

Developer Reference for Intel® oneAPI Math Kernel Library for C

Contents

Chapter 1: Developer Reference for Intel® oneAPI Math Kernel Library - C

Getting Help and Support	17
What's New	18
Notational Conventions	18
Overview	19
Performance Enhancements	22
Parallelism	23
C Datatypes Specific to Intel MKL	23
OpenMP* Offload	24
OpenMP* Offload for Intel® oneAPI Math Kernel Library	24
BLAS and Sparse BLAS Routines	32
BLAS Routines	32
Naming Conventions for BLAS Routines	32
C Interface Conventions for BLAS Routines	34
Matrix Storage Schemes for BLAS Routines	35
BLAS Level 1 Routines and Functions	35
BLAS Level 2 Routines	52
BLAS Level 3 Routines	96
Sparse BLAS Level 1 Routines	118
Vector Arguments	118
Naming Conventions for Sparse BLAS Routines	119
Routines and Data Types	119
BLAS Level 1 Routines That Can Work With Sparse Vectors	119
cblas_?axpyi	120
cblas_?doti	121
cblas_?dotci	121
cblas_?dotui	122
cblas_?gthr	123
cblas_?gthrz	124
cblas_?roti	124
cblas_?sctr	125
Sparse BLAS Level 2 and Level 3 Routines	126
Naming Conventions in Sparse BLAS Level 2 and Level 3	126
Sparse Matrix Storage Formats for Sparse BLAS Routines	127
Routines and Supported Operations	128
Interface Consideration	129
Sparse BLAS Level 2 and Level 3 Routines	133
Sparse QR Routines	240
mkl_sparse_set_qr_hint	240
mkl_sparse_?_qr	241
mkl_sparse_qr_reorder	243
mkl_sparse_?_qr_factorize	244
mkl_sparse_?_qr_solve	245
mkl_sparse_?_qr_qmult	247
mkl_sparse_?_qr_rsolve	249
Compact BLAS and LAPACK Functions	250
mkl_?gemm_compact	254

mkl_?trsm_compact.....	257
mkl_?potrf_compact.....	259
mkl_?getrfnp_compact	260
mkl_?geqrf_compact	261
mkl_?getrinp_compact	263
Numerical Limitations for Compact BLAS and Compact LAPACK	
Routines	264
mkl_?get_size_compact.....	265
mkl_get_format_compact	265
mkl_?gepack_compact	266
mkl_?geunpack_compact.....	268
Inspector-executor Sparse BLAS Routines.....	269
Naming Conventions in Inspector-Executor Sparse BLAS Routines	269
Sparse Matrix Storage Formats for Inspector-executor Sparse	
BLAS Routines	271
Supported Inspector-executor Sparse BLAS Operations	271
Two-stage Algorithm in Inspector-Executor Sparse BLAS Routines	272
Matrix Manipulation Routines	273
Inspector-Executor Sparse BLAS Analysis Routines	293
Inspector-Executor Sparse BLAS Execution Routines	310
BLAS-like Extensions	354
cblas_?axpy_batch.....	356
cblas_?axpy_batch_strided	357
cblas_?axpby	358
cblas_?gemmt.....	359
cblas_?gemm3m.....	362
cblas_?gemm_batch.....	366
cblas_?gemm_batch_strided	369
cblas_?gemm3m_batch_strided	372
cblas_?gemm3m_batch	376
cblas_?trsm_batch	379
cblas_?trsm_batch_strided.....	382
mkl_?imatcopy	384
mkl_?imatcopy_batch.....	386
mkl_?imatcopy_batch_strided	387
mkl_?omatadd_batch_strided.....	389
mkl_?omatcopy	391
mkl_?omatcopy_batch.....	393
mkl_?omatcopy_batch_strided	395
mkl_?omatcopy2	397
mkl_?omatadd	399
cblas_?gemm_pack_get_size, cblas_gemm_*_pack_get_size	401
cblas_?gemm_pack.....	403
cblas_gemm_*_pack	406
cblas_?gemm_compute	411
cblas_gemm_*_compute	414
cblas_gemm_bf16bf16f32_compute	420
cblas_gemm_bf16bf16f32.....	424
cblas_gemm_f16f16f32_compute.....	426
cblas_gemm_f16f16f32	430
cblas_?gemm_free.....	433
cblas_gemm_*	434
cblas_?gemv_batch_strided	438
cblas_?gemv_batch.....	439
cblas_?dgmm_batch_strided	441

cblas_?dggmm_batch.....	443
mkl_jit_create_?gemm	445
mkl_jit_get_?gemm_ptr	447
mkl_jit_destroy	450
LAPACK Routines.....	451
Choosing a LAPACK Routine.....	451
C Interface Conventions for LAPACK Routines.....	451
Matrix Layout for LAPACK Routines	453
Matrix Storage Schemes for LAPACK Routines	455
Mathematical Notation for LAPACK Routines	463
Error Analysis	464
LAPACK Linear Equation Routines	464
LAPACK Linear Equation Computational Routines	465
LAPACK Linear Equation Driver Routines	675
LAPACK Least Squares and Eigenvalue Problem Routines	779
LAPACK Least Squares and Eigenvalue Problem Computational Routines	780
LAPACK Least Squares and Eigenvalue Problem Driver Routines ..	1001
LAPACK Auxiliary Routines.....	1171
?lacgv	1171
?lacrm	1172
?syconv	1173
?syr	1174
i?max1	1176
?sum1	1176
?gelq2	1177
?geqr2	1178
?geqrt2	1180
?geqrt3	1182
?getf2	1184
?lacn2	1185
?lacpy	1187
?lakf2.....	1188
?lange	1189
?lansy	1190
?lanhe	1191
?lantr	1192
LAPACKE_set_nancheck.....	1194
LAPACKE_get_nancheck.....	1194
?lapmr	1194
?lapmt.....	1196
?lapy2	1197
?lapy3	1197
?laran.....	1198
?larfb	1198
?larfg	1201
?larft.....	1203
?larfx	1205
?large	1206
?larnd	1207
?larnv.....	1208
?laror	1209
?larot	1211
?lartgp	1214
?lartgs.....	1215

?lascl	1216
?lasd0	1217
?lasd1	1218
?lasd2	1221
?lasd3	1223
?lasd4	1225
?lasd5	1226
?lasd6	1227
?lasd7	1230
?lasd8	1233
?lasd9	1235
?lasda	1236
?lasdq	1239
?lasdt	1241
?laset	1241
?lasrt	1243
?laswp	1244
?latm1	1245
?latm2	1247
?latm3	1249
?latm5	1253
?latm6	1256
?latme	1258
?latmr	1262
?lauum	1268
?syswapr	1269
?heswapr	1270
?sfrk	1272
?hfrk	1273
?tfsn	1275
?tfttp	1277
?tfttr	1278
?tpqrt2	1280
?tprfb	1282
?tpttf	1285
?tpttr	1286
?trttf	1288
?trttp	1289
?lcp2	1290
?larcn	1291
mkl_?tpack	1292
mkl_?tpunpack	1294
LAPACK Utility Functions and Routines	1296
ilaver	1297
ilaenv	1297
?lamch	1300
LAPACK Test Functions and Routines	1301
?lagge	1301
?laghe	1302
?lagsy	1303
?latms	1304
Additional LAPACK Routines (Included for Compatibility with Netlib LAPACK)	1308
ScaLAPACK Routines	1312
Overview of ScaLAPACK Routines	1312

ScaLAPACK Array Descriptors.....	1313
Naming Conventions for ScaLAPACK Routines	1315
ScaLAPACK Computational Routines.....	1316
Systems of Linear Equations: ScaLAPACK Computational Routines.....	1316
Matrix Factorization: ScaLAPACK Computational Routines	1317
Solving Systems of Linear Equations: ScaLAPACK Computational Routines	1331
Estimating the Condition Number: ScaLAPACK Computational Routines	1347
Refining the Solution and Estimating Its Error: ScaLAPACK Computational Routines	1355
Matrix Inversion: ScaLAPACK Computational Routines.....	1365
Matrix Equilibration: ScaLAPACK Computational Routines	1369
Orthogonal Factorizations: ScaLAPACK Computational Routines..	1373
Symmetric Eigenvalue Problems: ScaLAPACK Computational Routines	1442
Nonsymmetric Eigenvalue Problems: ScaLAPACK Computational Routines	1475
Singular Value Decomposition: ScaLAPACK Driver Routines.....	1489
Generalized Symmetric-Definite Eigenvalue Problems: ScaLAPACK Computational Routines	1501
ScaLAPACK Driver Routines	1505
p?geevx	1505
p?gesv	1509
p?gesvx.....	1510
p?gbsv	1515
p?dbsv	1518
p?dtsv	1520
p?posv	1522
p?posvx.....	1524
p?pbsv	1529
p?ptsv	1531
p?gels	1533
p?syev	1536
p?syevd.....	1539
p?syevr	1541
p?syevx.....	1545
p?heev	1551
p?heevd	1554
p?heevr	1556
p?heevx	1561
p?gesvd.....	1568
p?sygvx.....	1572
p?hegvx	1579
ScaLAPACK Auxiliary Routines.....	1586
p?lacgv.....	1591
p?max1	1592
pilaver.....	1593
pmpcol	1594
pmpim2.....	1595
?combamax1.....	1596
p?sum1	1596
p?dbtrsv	1597
p?dttrsv.....	1600
p?gebal	1602

p?gebd2	1605
p?gehd2	1608
p?gelq2	1610
p?geql2	1612
p?geqr2	1614
p?gerq2	1616
p?getf2	1618
p?labrd	1620
p?lacon	1623
p?laconsb	1625
p?lACP2	1626
p?lACP3	1627
p?lACP	1629
p?laevswp	1630
p?lahrd	1632
p?laiect	1634
p?lamve	1635
p?lange	1636
p?lanhs	1638
p?lansy, p?lanhe	1640
p?lantr	1642
p?lapiv	1643
p?lapv2	1646
p?laqge	1648
p?laqr0	1649
p?laqr1	1652
p?laqr2	1655
p?laqr3	1657
p?laqr5	1660
p?laqsy	1662
p?lared1d	1664
p?lared2d	1665
p?larf	1666
p?larfb	1669
p?larfc	1672
p?larfg	1674
p?larft	1676
p?larz	1678
p?larzb	1681
p?larzc	1685
p?larzt	1687
p?lascl	1690
p?lase2	1692
p?laset	1693
p?lasmsub	1695
p?lasrt	1696
p?lassq	1698
p?laswp	1699
p?latra	1701
p?latrd	1702
p?latrs	1705
p?latrz	1707
p?lauu2	1709
p?lauum	1711
p?lawil	1712

p?org2l/p?ung2l.....	1713
p?org2r/p?ung2r.....	1715
p?orgl2/p?ungl2.....	1717
p?orgr2/p?ungr2.....	1719
p?orm2l/p?unm2l.....	1721
p?orm2r/p?unm2r.....	1724
p?orml2/p?unml2.....	1728
p?ormr2/p?unmr2.....	1731
p?pbtrsv	1734
p?pttrsv	1738
p?potf2.....	1740
p?rot.....	1742
p?rscl	1744
p?sygs2/p?hegs2	1745
p?sytd2/p?hetd2.....	1747
p?trord	1750
p?trsen.....	1754
p?trti2	1758
?lahqr2.....	1759
?lamsh	1761
?lapst.....	1762
?laqr6	1763
?lar1va	1766
?laref	1767
?larrb2	1770
?larrd2	1772
?larre2	1775
?larre2a.....	1778
?larrf2	1782
?larrv2	1783
?lasorte	1788
?lasrt2.....	1789
?stegr2.....	1790
?stegr2a	1793
?stegr2b.....	1796
?stein2	1799
?dbtf2	1801
?dbtrf.....	1802
?dttrf	1803
?dttrsv	1804
?pttrsv	1806
?steqr2.....	1807
?trmvt.....	1809
pilaenv	1811
pilaenvx	1812
pjlaenv	1814
Additional ScaLAPACK Routines.....	1815
ScaLAPACK Utility Functions and Routines	1817
p?labad	1818
p?lachkieee.....	1819
p?lamch	1819
p?lasnbt	1820
descinit	1821
numroc	1822
ScaLAPACK Redistribution/Copy Routines	1823

p?gemr2d	1823
p?trmr2d	1825
Sparse Solver Routines	1827
oneMKL PARDISO - Parallel Direct Sparse Solver Interface	1827
pardiso	1834
pardisoinit	1841
pardiso_64	1842
mkl_pardiso_pivot	1843
pardiso_getdiag	1843
pardiso_export	1845
pardiso_handle_store	1847
pardiso_handle_restore	1848
pardiso_handle_delete	1848
pardiso_handle_store_64	1849
pardiso_handle_restore_64	1850
pardiso_handle_delete_64	1851
oneMKL PARDISO Parameters in Tabular Form	1851
pardiso iparm Parameter	1856
PARDISO_DATA_TYPE	1869
Parallel Direct Sparse Solver for Clusters Interface	1870
cluster_sparse_solver	1872
cluster_sparse_solver_64	1876
cluster_sparse_solver_get_csr_size	1877
cluster_sparse_solver_set_csr_ptrs	1879
cluster_sparse_solver_set_ptr	1881
cluster_sparse_solver_export	1883
cluster_sparse_solver iparm Parameter	1884
Direct Sparse Solver (DSS) Interface Routines	1893
DSS Interface Description	1895
DSS Implementation Details	1896
DSS Routines	1897
Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)	1908
CG Interface Description	1909
FGMRES Interface Description	1914
RCI ISS Routines	1920
RCI ISS Implementation Details	1933
Preconditioners based on Incomplete LU Factorization Technique	1934
ILU0 and ILUT Preconditioners Interface Description	1934
dcsrilu0	1935
dcsrilit	1938
Sparse Matrix Checker Routines	1941
sparse_matrix_checker	1941
sparse_matrix_checker_init	1943
Extended Eigensolver Routines	1944
The FEAST Algorithm	1944
Extended Eigensolver Functionality	1946
Parallelism in Extended Eigensolver Routines	1947
Achieving Performance With Extended Eigensolver Routines	1947
Extended Eigensolver Interfaces for Eigenvalues within Interval	1948
Extended Eigensolver Naming Conventions	1948
feastinit	1949
Extended Eigensolver Input Parameters	1949
Extended Eigensolver Output Details	1951
Extended Eigensolver RCI Routines	1952

Extended Eigensolver Predefined Interfaces.....	1957
Extended Eigensolver Interfaces for Extremal Eigenvalues/Singular Values	1970
Extended Eigensolver Interfaces to find largest/smallest eigenvalues.....	1970
Extended Eigensolver Interfaces to find largest/smallest singular values	1975
mkl_sparse_ee_init	1977
Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem.....	1977
Vector Mathematical Functions	1979
VM Data Types, Accuracy Modes, and Performance Tips.....	1980
VM Naming Conventions	1980
VM Function Interfaces	1981
Vector Indexing Methods.....	1983
VM Error Diagnostics	1984
VM Mathematical Functions	1985
Special Value Notations.....	1987
Arithmetic Functions.....	1988
Power and Root Functions	2005
Exponential and Logarithmic Functions	2023
Trigonometric Functions	2038
Hyperbolic Functions	2068
Special Functions	2081
Rounding Functions	2097
VM Pack/Unpack Functions	2108
v?Pack	2108
v?Unpack.....	2109
VM Service Functions.....	2110
vmlSetMode	2111
vmlGetMode.....	2113
MKLFreeTls	2113
vmlSetErrStatus	2114
vmlGetErrStatus	2115
vmlClearErrStatus.....	2115
vmlSetErrorCallBack.....	2116
vmlGetErrorCallBack	2118
vmlClearErrorCallBack	2118
Miscellaneous VM Functions	2118
v?CopySign	2118
v?NextAfter.....	2119
v?Fdim	2121
v?Fmax	2122
v?Fmin	2123
v?MaxMag.....	2124
v?MinMag	2126
Statistical Functions.....	2127
Random Number Generators.....	2128
Random Number Generators Conventions	2128
Basic Generators.....	2134
Error Reporting.....	2137
VS RNG Usage ModelIntel® oneMKL RNG Usage Model.....	2139
Service Routines	2140
Distribution Generators.....	2161
Advanced Service Routines.....	2206

Convolution and Correlation.....	2211
Convolution and Correlation Naming Conventions.....	2212
Convolution and Correlation Data Types.....	2213
Convolution and Correlation Parameters.....	2213
Convolution and Correlation Task Status and Error Reporting	2215
Convolution and Correlation Task Constructors.....	2216
Convolution and Correlation Task Editors.....	2223
Task Execution Routines.....	2228
Convolution and Correlation Task Destructors	2235
Convolution and Correlation Task Copiers	2236
Convolution and Correlation Usage Examples.....	2237
Convolution and Correlation Mathematical Notation and Definitions	2241
Convolution and Correlation Data Allocation.....	2242
Summary Statistics	2244
Summary Statistics Naming Conventions	2245
Summary Statistics Data Types.....	2245
Summary Statistics Parameters	2246
Summary Statistics Task Status and Error Reporting	2246
Summary Statistics Task Constructors	2250
Summary Statistics Task Editors	2252
Summary Statistics Task Computation Routines	2279
Summary Statistics Task Destructor	2284
Summary Statistics Usage Examples	2284
Summary Statistics Mathematical Notation and Definitions	2286
Fourier Transform Functions.....	2290
FFT Functions	2291
FFT Interface.....	2292
Computing an FFT.....	2292
Configuration Settings	2293
FFT Descriptor Manipulation Functions	2308
FFT Descriptor Configuration Functions	2312
FFT Computation Functions	2314
Status Checking Functions	2321
Cluster FFT Functions	2323
Computing Cluster FFT	2324
Distributing Data Among Processes	2325
Cluster FFT Interface.....	2326
Cluster FFT Descriptor Manipulation Functions.....	2327
Cluster FFT Computation Functions.....	2329
Cluster FFT Descriptor Configuration Functions.....	2332
Error Codes.....	2336
PBLAS Routines.....	2336
PBLAS Routines Overview.....	2337
PBLAS Routine Naming Conventions	2338
PBLAS Level 1 Routines.....	2339
p?amax	2340
p?asum	2341
p?axpy	2342
p?copy	2343
p?dot	2344
p?dotc.....	2346
p?dotu.....	2347
p?nrm2	2348
p?scal	2349

p?swap.....	2350
PBLAS Level 2 Routines.....	2351
p?gemv	2352
p?agemv	2354
p?ger	2357
p?gerc.....	2358
p?geru	2360
p?hemv	2362
p?ahemv	2363
p?her	2365
p?her2	2367
p?symv	2369
p?asymv.....	2371
p?syr	2372
p?syr2.....	2374
p?trmv	2376
p?atrmv.....	2378
p?trsv	2380
PBLAS Level 3 Routines.....	2382
p?geadd	2383
p?tradd	2384
p?gemm	2386
p?hemm	2388
p?herk.....	2390
p?her2k.....	2392
p?symm	2394
p?syrk	2396
p?syr2k	2398
p?tran	2401
p?tranu	2402
p?tranc.....	2403
p?trmm	2404
p?trsm	2407
Partial Differential Equations Support	2409
Trigonometric Transform Routines.....	2409
Trigonometric Transforms Implemented	2410
Sequence of Invoking TT Routines	2411
Trigonometric Transform Interface Description	2412
TT Routines.....	2413
Common Parameters of the Trigonometric Transforms.....	2420
Trigonometric Transform Implementation Details	2423
Fast Poisson Solver Routines	2424
Poisson Solver Implementation	2424
Sequence of Invoking Poisson Solver Routines	2430
Fast Poisson Solver Interface Description	2432
Routines for the Cartesian Solver	2433
Routines for the Spherical Solver	2442
Common Parameters for the Poisson Solver.....	2449
Poisson Solver Implementation Details.....	2458
Nonlinear Optimization Problem Solvers	2459
Nonlinear Solver Organization and Implementation	2459
Nonlinear Solver Routine Naming Conventions	2461
Nonlinear Least Squares Problem without Constraints	2461
?trnlsp_init	2462
?trnlsp_check	2464

?trnlsolve.....	2465
?trnlsolve_get	2467
?trnlsolve_delete	2468
Nonlinear Least Squares Problem with Linear (Bound) Constraints	2469
?trnlsolve_init	2469
?trnlsolve_check.....	2471
?trnlsolve_solve.....	2473
?trnlsolve_get	2474
?trnlsolve_delete	2476
Jacobian Matrix Calculation Routines	2476
?jacobi_init	2477
?jacobi_solve	2478
?jacobi_delete	2479
?jacobi	2479
?jacobix.....	2480
Support Functions	2482
Version Information	2485
mkl_get_version	2485
mkl_get_version_string	2487
Threading Control	2487
mkl_set_num_threads.....	2488
mkl_domain_set_num_threads.....	2489
mkl_set_num_threads_local	2490
mkl_set_dynamic.....	2492
mkl_get_max_threads.....	2493
mkl_domain_get_max_threads.....	2494
mkl_get_dynamic	2495
mkl_set_num_stripes	2496
mkl_get_num_stripes.....	2496
Error Handling	2497
Error Handling for Linear Algebra Routines	2497
Handling Fatal Errors	2500
Character Equality Testing	2501
lsame	2501
lsamen	2502
Timing	2502
second/dsecond	2502
mkl_get_cpu_clocks.....	2503
mkl_get_cpu_frequency.....	2504
mkl_get_max_cpu_frequency	2504
mkl_get_clocks_frequency	2505
Memory Management	2505
mkl_free_buffers	2505
mkl_thread_free_buffers.....	2506
mkl_disable_fast_mm	2507
mkl_mem_stat	2507
mkl_peak_mem_usage	2508
mkl_malloc	2509
mkl_calloc	2510
mkl_realloc	2511
mkl_free.....	2511
mkl_set_memory_limit	2512
Usage Examples for the Memory Functions.....	2513
Single Dynamic Library Control	2514
mkl_set_interface_layer.....	2514

mkl_set_threading_layer	2515
mkl_set_xerbla.....	2516
mkl_set_progress	2517
mkl_set_pardiso_pivot.....	2517
Conditional Numerical Reproducibility Control.....	2518
mkl_cbwr_set.....	2519
mkl_cbwr_get	2520
mkl_cbwr_get_auto_branch	2521
Named Constants for CNR Control	2521
Reproducibility Conditions	2523
Usage Examples for CNR Support Functions.....	2523
Miscellaneous	2524
mkl_progress	2524
mkl_enable_instructions	2526
mkl_set_env_mode.....	2528
mkl_verbose	2529
mkl_verbose_output_file.....	2530
mkl_set_mpi	2531
mkl_finalize	2532
BLACS Routines	2533
Matrix Shapes.....	2533
Repeatability and Coherence.....	2535
BLACS Combine Operations	2537
?gamx2d	2538
?gamn2d	2540
?gsum2d	2541
BLACS Point To Point Communication	2542
?gesd2d	2544
?trsd2d.....	2545
?gerv2d	2545
?trrv2d	2546
BLACS Broadcast Routines.....	2547
?gebs2d	2548
?trbs2d.....	2548
?gebr2d	2549
?trbr2d	2550
BLACS Support Routines	2551
Initialization Routines	2551
Destruction Routines	2558
Informational Routines	2559
Miscellaneous Routines	2561
BLACS Routines Usage Examples.....	2562
Data Fitting Functions	2562
Data Fitting Function Naming Conventions.....	2562
Data Fitting Function Data Types	2563
Mathematical Conventions for Data Fitting Functions.....	2563
Data Fitting Usage Model.....	2566
Data Fitting Usage Examples	2566
Data Fitting Function Task Status and Error Reporting	2572
Data Fitting Task Creation and Initialization Routines	2574
df?NewTask1D	2574
Task Configuration Routines.....	2576
df?EditPPSpline1D	2577
df?EditPtr	2584

dfiEditVal	2585
df?EditIdxPtr.....	2587
df?QueryPtr.....	2589
dfiQueryVal.....	2589
df?QueryIdxPtr.....	2590
Data Fitting Computational Routines	2591
df?Construct1D.....	2592
df?Interpolate1D/df?InterpolateEx1D	2593
df?Integrate1D/df?IntegrateEx1D	2601
df?SearchCells1D/df?SearchCellsEx1D	2605
df?InterpCallBack	2607
df?IntegrCallBack	2608
df?SearchCellsCallBack	2610
Data Fitting Task Destructors	2611
dfDeleteTask	2611
Appendix A: Linear Solvers Basics	2612
Sparse Linear Systems.....	2612
Matrix Fundamentals	2613
Direct Method.....	2614
Sparse Matrix Storage Formats	2620
DSS Symmetric Matrix Storage	2621
DSS Nonsymmetric Matrix Storage.....	2622
DSS Structurally Symmetric Matrix Storage.....	2622
DSS Distributed Symmetric Matrix Storage.....	2623
Sparse BLAS CSR Matrix Storage Format.....	2624
Sparse BLAS CSC Matrix Storage Format.....	2626
Sparse BLAS Coordinate Matrix Storage Format	2627
Sparse BLAS Diagonal Matrix Storage Format	2628
Sparse BLAS Skyline Matrix Storage Format	2629
Sparse BLAS BSR Matrix Storage Format.....	2630
Appendix B: Routine and Function Arguments	2632
Vector Arguments in BLAS.....	2632
Vector Arguments in Vector Math	2633
Matrix Arguments.....	2634
Appendix C: FFTW Interface to Intel® oneAPI Math Kernel Library (oneMKL) ..	2639
Notational Conventions	2639
FFTW2 Interface to Intel® oneAPI Math Kernel Library (oneMKL)	2639
Wrappers Reference	2640
Limitations of the FFTW2 Interface to Intel® oneAPI Math Kernel Library (oneMKL)	2642
Installing FFTW2 Interface Wrappers	2643
MPI FFTW2 Wrappers	2644
FFTW3 Interface to Intel® oneAPI Math Kernel Library (oneMKL)	2647
Using FFTW3 Wrappers.....	2647
Building Your Own Wrapper Library.....	2649
Building an Application With FFTW3 Interface Wrappers	2649
Running FFTW3 Interface Wrapper Examples	2650
MPI FFTW3 Wrappers	2650
Appendix D: Code Examples	2651
BLAS Code Examples.....	2651
Fourier Transform Functions Code Examples	2657
FFT Code Examples.....	2657
Examples for Cluster FFT Functions	2663
Auxiliary Data Transformations	2665

Appendix F: oneMKL Functionality.....	2666
BLAS Functionality	2666
Transposition Functionality	2667
LAPACK Functionality	2667
DFT Functionality	2668
Sparse BLAS Functionality	2669
Sparse Solvers Functionality	2674
Random Number Generators Functionality	2674
Vector Math Functionality	2676
Data Fitting Functionality	2676
Summary Statistics Functionality	2677
Bibliography	2678
Glossary.....	2683
Notices and Disclaimers.....	2688

Developer Reference for Intel® oneAPI Math Kernel Library - C

1

For more documentation on this and other products, visit the [oneAPI Documentation Library](#).

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

Documentation for versions of Intel® Math Kernel Library older than 2023.0 is available for download only. See [Downloadable Documentation](#).

[What's New](#)

[Fortran interface: Developer Reference for Intel® oneAPI Math Kernel Library - Fortran](#)

This publication describes the C interface.

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

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at [www.Intel.com/PerformanceIndex](http://www.intel.com/PerformanceIndex).

Notice revision #20201201

Getting Help and Support

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

What's New

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

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

- [Download Documentation for Intel® Parallel Studio XE](#)
- [Download Documentation for Intel® System Studio](#)

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

Product and Performance Information

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

Notice revision #20201201

Notational Conventions

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

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

This manual uses the following notational conventions:

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

Routine Name Shorthand

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

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

Font Conventions

The following font conventions are used:

<code>lowercase courier</code>	Code examples:
	<code>a[k+i][j] = matrix[i][j];</code>
	data types; for example, <code>const float*</code>

<code>lowercase courier mixed with UpperCase courier</code>	Function names; for example, <code>vmlSetMode</code>
<code>lowercase courier italic</code>	Variables in arguments and parameters description. For example, <i>incx</i> .
<code>*</code>	Used as a multiplication symbol in code examples and equations and where required by the programming language syntax.

Overview

NOTE

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

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

NOTE

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

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

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

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

NOTE

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

BLAS Routines

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

- **BLAS Level 1 Routines** perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.

- [BLAS Level 2 Routines](#) perform matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
- [BLAS Level 3 Routines](#) perform matrix-matrix operations, such as matrix-matrix multiplication, rank-k update, and solution of triangular systems.

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

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

Sparse BLAS Routines

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

Sparse QR

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

LAPACK Routines

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

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

The [LAPACK routines](#) can be divided into the following groups according to the operations they perform:

- Routines for solving systems of linear equations, factoring and inverting matrices, and estimating condition numbers (see [LAPACK Routines: Linear Equations](#)).
- Routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations (see [LAPACK Routines: Least Squares and Eigenvalue Problems](#)).

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

Sparse Solver Routines

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

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

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

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

Extended Eigensolver Routines

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

It is free from orthogonalization procedures. Its main computational tasks consist of solving very few inner independent linear systems with multiple right-hand sides and one reduced eigenvalue problem orders of magnitude smaller than the original one. The Feast algorithm combines simplicity and efficiency and offers many important capabilities for achieving high performance, robustness, accuracy, and scalability on parallel architectures. This algorithm is expected to significantly augment numerical performance in large-scale modern applications.

Some of the characteristics of the Feast algorithm [\[Polizzi09\]](#) are:

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

This capability can also be used for solving a series of eigenvalue problems that are close one another.

- The number of internal iterations is independent of the size of the system and the number of eigenpairs in the search interval
- The inner linear systems can be solved either iteratively (even with modest relative residual error) or directly

VM Functions

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

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

Statistical Functions

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

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

Fourier Transform Functions

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

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

Partial Differential Equations Support

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

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

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

Support Functions

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

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

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

Product and Performance Information

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

Notice revision #20201201

Performance Enhancements

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

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

The major optimization techniques used throughout the library include:

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

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

Product and Performance Information

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

Notice revision #20201201

Parallelism

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

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

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

Product and Performance Information

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

Notice revision #20201201

C Datatypes Specific to Intel MKL

The `mk1_types.h` file defines datatypes specific to Intel® oneAPI Math Kernel Library (oneMKL).

C/C++ Type	Fortran Type	LP32 Equivalent (Size in Bytes)	LP64 Equivalent (Size in Bytes)	ILP64 Equivalent (Size in Bytes)
MKL_INT (MKL integer)	INTEGER (default INTEGER)	C/C++: int Fortran: INTEGER*4	C/C++: int Fortran: INTEGER*4 (4 bytes)	C/C++: long long (or define MKL_ILP64 macros Fortran: INTEGER*8

C/C++ Type	Fortran Type	LP32 Equivalent (Size in Bytes)	LP64 Equivalent (Size in Bytes)	ILP64 Equivalent (Size in Bytes)
MKL_UINT (MKL unsigned integer)	N/A	(4 bytes) C/C++: unsigned int	C/C++: unsigned int (4 bytes)	C/C++: unsigned long long (8 bytes)
MKL_LONG (MKL long integer)	N/A	(4 bytes) C/C++: long	C/C++: long (Windows: 4 bytes) (Linux, Mac: 8 bytes)	C/C++: long (8 bytes)
MKL_Complex8 (Like C99 complex float)	COMPLEX*8	(8 bytes)	(8 bytes)	(8 bytes)
MKL_Complex16 (Like C99 complex double)	COMPLEX*16	(16 bytes)	(16 bytes)	(16 bytes)

You can redefine datatypes specific to Intel® oneAPI Math Kernel Library (oneMKL). One reason to do this is if you have your own types which are binary-compatible with Intel® oneAPI Math Kernel Library (oneMKL) datatypes, with the same representation or memory layout. To redefine a datatype, use one of these methods:

- Insert the `#define` statement redefining the datatype before the `mkl.h` header file `#include` statement. For example,

```
#define MKL_INT size_t
#include "mkl.h"
```

- Use the compiler `-D` option to redefine the datatype. For example,

```
...-DMKL_INT=size_t...
```

NOTE

As the user, if you redefine Intel® oneAPI Math Kernel Library (oneMKL) datatypes you are responsible for making sure that your definition is compatible with that of Intel® oneAPI Math Kernel Library (oneMKL). If not, it might cause unpredictable results or crash the application.

OpenMP* Offload

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

OpenMP* Offload for Intel® oneAPI Math Kernel Library

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

- All Level 1, 2, and 3 BLAS functions through the CBLAS and BLAS interfaces, supporting both synchronous and asynchronous execution

- **BLAS-like extensions:** `cblas_?axpby`, `cblas_?axpy_batch{_strided}`, `cblas_?copy_batch{_strided}`, `cblas_?gemv_batch{_strided}`, `cblas_?dgmm_batch{_strided}`, `cblas_hgemm`, `cblas_gemm_bf16bf16f32`, `cblas_gemm_s8u8s32`, `cblas_?gemm_batch{_strided}`, `cblas_?trsm_batch{_strided}`, and `cblas_?gemmt` functionality through the CBLAS and BLAS interfaces as well as `mkl_?omatcopy_batch_strided`, `mkl_?imatcopy_batch_strided`, and `mkl_?omatadd_batch_strided`, supporting both synchronous and asynchronous execution
- LAPACK, including LAPACK-like extensions
 - All computations on the Intel GPU (supports both synchronous and asynchronous execution):
 - `?getrf_batch`
 - `?getrf_batch_strided`
 - `?getrfnp_batch`
 - `?getrfnp_batch_strided`
 - `?getri`
 - `?getri_oop_batch`
 - `?getri_oop_batch_strided`
 - `?getrs`
 - `?getrs_batch_strided`
 - `?getrsnp_batch_strided`
 - `?gels_batch_strided`
 - `?potrf`
 - `?potri`
 - `?potrs`
 - `?trtri`
 - `?trtrs`
 - Hybrid; some computations on the Intel GPU (supports synchronous execution):
 - `?geqrf`
 - `?getrf` (all computations on the CPU for $n \leq 256$)
 - `mkl_?getrfnp` (all computations on the CPU for $n \leq 512$)
 - `?ormqr`, `?unmqr`
 - Interface support only; all computations on the CPU (supports synchronous execution):
 - `?gebrd`
 - `?gesvd`
 - `?orgqr`, `?ungqr`
 - `?steqr`
 - `?syev`, `?heev`
 - `?syevd`, `?heevd`
 - `?syevx`, `?heevx`
 - `?sygvd`, `?hegvd`
 - `?sygvx`, `?hegvx`
 - `?sytrd`, `?hetrd`
- Vector Statistics
 - Random number generators

NOTE

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

Basic random number generators:

- `VSL_BRNG_MCG31`
- `VSL_BRNG_MCG59`
- `VSL_BRNG_PHILOX4X32X10`
- `VSL_BRNG_MRG32K3A`
- `VSL_BRNG_MT19937`

- VSL_BRNG_MT2203
- VSL_BRNG_SOBOL

Important Check the [oneMKL DPC++ developer reference](#) for the BRNG data type used in the distributions in case the offload device doesn't have `sycl::aspect::fp64` support.

- Summary statistics

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

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

Supported methods:

- VSL_SS_METHOD_FAST
- VSL_SS_METHOD_FAST_USER_MEAN

- FFTs through both DFTI and FFTW3 interfaces in one, two, and three dimensions.
 - For COMPLEX_STORAGE, only the DFTI_COMPLEX_COMPLEX format is currently supported on CPU and GPU devices.
 - Both synchronous and asynchronous computations are supported.
 - For R2C/C2R transforms on the GPU, only DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX is supported (implying DFTI_PACKED_FORMAT=DFTI_CCE_FORMAT).
- **NOTE** `INCONSISTENT_CONFIGURATION` errors at compute time indicate an invalid descriptor or invalid data pointer. Double check your data mapping if you encounter such errors.
- Arbitrary strides and batch distances are not supported for multi-dimensional R2C transforms offloaded to the GPU. Considering the last dimension of the data, every element must be separated from its two nearest peers (along another dimension and/or in another batch) by a constant distance. For example, to compute a batched, two-dimensional R2C FFT of size `[N2, N1]` with input strides `[0, S2, 1]` (row-major layout with unit elementary stride and no offset), `INPUT_DISTANCE` must be equal to `N2*S2` so that every element is separated from its nearest last-dimension counterpart(s) by a distance `S2` (in this example), even across batches.
- Due to the variadic implementation of `DftiComputeForward` and `DftiComputeBackward`, out-of-place compute calls using the DFTI API with the OpenMP 5.1 dispatch construct differ from common dispatch construct usage by requiring a "need_device_ptr" clause. The oneMKL examples provided on installation demonstrate this usage.
- Transforms on GPU devices may overwrite FFT-irrelevant, padding entries in the output data.
- Sparse BLAS
 - `mkl_sparse_{s, d}_create_csr`
 - `mkl_sparse_{s, d}_export_csr`

- `mkl_sparse_destroy`
- `mkl_sparse_order`
 - Currently supports only CSR matrix format.
- `mkl_sparse_set_mv_hint`
 - Currently supports only `SPARSE_OPERATION_NON_TRANSPOSE` with CSR matrix format for general MV (`SPARSE_MATRIX_TYPE_GENERAL`) and triangular MV (`SPARSE_MATRIX_TYPE_TRIANGULAR` with fill modes `SPARSE_FILL_MODE_LOWER/SPARSE_FILL_MODE_UPPER`).
- `mkl_sparse_set_sv_hint`
- `mkl_sparse_optimize`
 - Supports optimization for `mkl_sparse_{s, d}_mv` functionality based on supported hints added through `mkl_sparse_set_mv_hint` offload.
 - Supports optimization for `mkl_sparse_{s, d}_trsv` functionality based on supported hints added through `mkl_sparse_set_sv_hint` offload.
 - Both synchronous and asynchronous executions are supported.

NOTE Note that although you can run the `mkl_sparse_optimize` offload function asynchronously, you are responsible for the data dependency between the optimization routine and the execution routines.

- `mkl_sparse_{s, d}_mv`:
 - Currently supports only `SPARSE_OPERATION_NON_TRANSPOSE` with the following combinations of matrix types:
 - `SPARSE_MATRIX_TYPE_GENERAL`
 - `SPARSE_MATRIX_TYPE_TRIANGULAR` with fill modes `SPARSE_FILL_MODE_LOWER/SPARSE_FILL_MODE_UPPER` and diagonal types `SPARSE_DIAG_UNIT/SPARSE_DIAG_NON_UNIT`
 - `SPARSE_MATRIX_TYPE_SYMMETRIC` fill modes `SPARSE_FILL_MODE_LOWER/SPARSE_FILL_MODE_UPPER` and diagonal type `SPARSE_DIAG_NON_UNIT` (currently, `SPARSE_DIAG_UNIT` is not supported)
 - Both synchronous and asynchronous computations are supported.
- `mkl_sparse_{s, d}_mm`:
 - Currently supported only with `SPARSE_MATRIX_TYPE_GENERAL` and `SPARSE_OPERATION_NON_TRANSPOSE`.
 - Both `SPARSE_LAYOUT_ROW_MAJOR` and `SPARSE_LAYOUT_COLUMN_MAJOR` are supported.
 - Both synchronous and asynchronous computations are supported.
- `mkl_sparse_{s, d}_trsv`
 - Currently only supported with `SPARSE_MATRIX_TYPE_TRIANGULAR`, `SPARSE_OPERATION_NON_TRANSPOSE`, and `alpha == 1.0`
 - Both synchronous and asynchronous computations are supported
- `mkl_sparse_sp2m`
 - Currently supported only with `SPARSE_MATRIX_TYPE_GENERAL`.
 - Both synchronous and asynchronous computations are supported with Level Zero backend, and currently only synchronous computations are supported with OpenCL backend.
 - Note that you can run the `mkl_sparse_sp2m` offload function asynchronously, but you are responsible for the data dependency between the first stage and the second stage of `mkl_sparse_sp2m`.
 - `mkl_sparse_sp2m` internally creates arrays for the sparse C matrix output. As they may be expected to be used subsequently on both host and device, they are created internally using USM shared memory. The arrays are managed by the library and will be cleaned up when the corresponding C matrix handle is destroyed; however, direct access to the arrays is provided by the `mkl_sparse_{s,d}_export_csr()` OpenMP offload function. Users are recommended to make a copy to their own arrays if they want to have such data beyond the scope of the C matrix handle.

The choice of USM shared memory for C arrays is made for functional support of the OpenMP Offload paradigm and has a performance impact over choosing USM device memory, which would be more performant but not functional in all subsequent use cases.

- The created C matrix in the provided handle is not guaranteed to be sorted, so the `mkl_sparse_order()` OpenMP offload API is provided for user convenience if that property is needed.
- The input matrix handle A is not required to be sorted on input, but the input matrix handle B is required to be sorted on input.
- In Sparse BLAS, the usage model consists of the creation stage, the inspection stage, the execution stage, and the destruction stage. For Sparse BLAS with C OpenMP Offload, all stages can be asynchronously executed, provided any data dependencies are already respected.

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

```
void cblas_sgemm(const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE TransA,
    const CBLAS_TRANSPOSE TransB, const MKL_INT M, const MKL_INT N,
    const MKL_INT K, const float alpha, const float *A, const MKL_INT lda,
    const float *B, const MKL_INT ldb, const float beta, float *C,
    const MKL_INT ldc);
```

If the oneMKL function (for example, `cblas_sgemm`) is called outside of an `omp target variant dispatch` section or if offload is disabled, then the CPU implementation is dispatched. If the same function is called within an `omp target variant dispatch` section and offload is possible then the GPU implementation is dispatched. By default the execution of the oneMKL function within a dispatch variant construct is synchronous. OpenMP offload computations may be done asynchronously by adding the `nowait` clause to the target variant dispatch construct. This ensures that the host thread encountering the task region generated by this target construct will not be blocked by the oneMKL call. Rather, the host thread is returned to the caller for further use. To finish the asynchronous (`nowait`) computations and ensure memory and execution model consistency (for example, that the results of a computation will be ready in memory to map), the last such `nowait` computation is followed by the stand-alone construct `#pragma omp taskwait`.

From the *OpenMP Application Programming Interface* version 5.0 specification: "The taskwait region binds to the current task region [i.e., in this case, the last `nowait` computation]. The current task region is suspended at an implicit task scheduling point associated with the construct. The current task region remains suspended until all child tasks that it generated before the taskwait region complete execution [currently, depend clause is not supported]."

Example

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

```
examples/c_offload
```

The following code snippet shows how to use OpenMP offload for single-call oneMKL features such as most dense linear algebra functionality.

```
#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h" // MKL header file for OpenMP offload
int dnum = 0;
int main() {
    float *a, *b, *c, alpha = 1.0, beta = 1.0;
    MKL_INT m = 150, n = 200, k = 128, lda = m, ldb = k, ldc = m;
    MKL_INT sizea = lda * k, sizeb = ldb * n, sizec = ldc * n;
    // allocate matrices and check pointers
    a = (float *)mkl_malloc(sizea * sizeof(float), 64);
    ...
    // initialize matrices
    #pragma omp target map(c[0:sizec])
```



```

{
    for (i = 0; i < sizec; i++) {
        c[i] = 42;
    }
    ...
}
// run gemm on host, use standard MKL interface
cblas_sgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha, a,
lda, b, ldb, beta, c, ldc);
// map the a, b, and c matrices on the device memory
#pragma omp target data map(to:a[0:sizea],b[0:sizeb]) map(tofrom:c[0:sizec])
device(dnum)
{
    // run gemm on gpu, use standard MKL interface within a variant dispatch
construct
    // if offload is not possible, default to cpu
    // use the use_device_ptr clause to specify that a, b and c are device
memory
#pragma omp target variant dispatch device(dnum) use_device_ptr(a, b, c)
{
    cblas_sgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k,
alpha, a, lda, b, ldb, beta, c, ldc);
}
}
// Free matrices
mkl_free(a);
...
}

```

Some of the oneMKL functionality requires to call a set of functions to perform the corresponding computation. This is the case, for example, for the Discrete Fourier Transform which for a typical computation involves calling the functions.

```

DFTI_EXTERN MKL_LONG DftiCreateDescriptor(DFTI_DESCRIPTOR_HANDLE*,
enum DFTI_CONFIG_VALUE,
enum DFTI_CONFIG_VALUE,
MKL_LONG, ...);
DFTI_EXTERN MKL_LONG DftiCommitDescriptor(DFTI_DESCRIPTOR_HANDLE);
DFTI_EXTERN MKL_LONG DftiComputeForward(DFTI_DESCRIPTOR_HANDLE, void*, ...);
DFTI_EXTERN MKL_LONG DftiComputeBackward(DFTI_DESCRIPTOR_HANDLE, void*, ...);
DFTI_EXTERN MKL_LONG DftiFreeDescriptor(DFTI_DESCRIPTOR_HANDLE*);

```

In that case, only a subset of the calls must be wrapped in an `omp target variant dispatch construct` as shown in the following code snippet for DFTI.

```

#include <omp.h>
#include "mkl.h"
#include "mkl_omp_offload.h"
int main(void)
{
    const int devNum = 0;
    const MKL_LONG N = 64;        // Size of 1D transform
    MKL_LONG status = 0;
    MKL_LONG statusGPU = 0;
    DFTI_DESCRIPTOR_HANDLE descHandle = NULL;
    DFTI_DESCRIPTOR_HANDLE descHandleGPU = NULL;
    MKL_Complex8 *x = NULL;
    MKL_Complex8 *xGPU = NULL;
    printf("Create DFTI descriptor\n");

```

```

    status = DftiCreateDescriptor(&descHandle, DFTI_SINGLE, DFTI_COMPLEX, 1, N);
    printf("Create GPU DFTI descriptor\n");
    statusGPU = DftiCreateDescriptor(&descHandleGPU, DFTI_SINGLE, DFTI_COMPLEX,
1, N);
    printf("Commit DFTI descriptor\n");
    status = DftiCommitDescriptor(descHandle);
    printf("Commit GPU DFTI descriptor\n");
#pragma omp target variant dispatch device(devNum)
    {
        statusGPU = DftiCommitDescriptor(descHandleGPU);
    }
    printf("Allocate memory for input array\n");
    x = (MKL_Complex8 *)mkl_malloc(N*sizeof(MKL_Complex8), 64);
    printf("Allocate memory for GPU input array\n");
    xGPU = (MKL_Complex8 *)mkl_malloc(N*sizeof(MKL_Complex8), 64);
    printf("Initialize input for forward FFT\n");
    // init x and xGPU ...
    printf("Compute forward FFT in-place\n");
    status = DftiComputeForward(descHandle, x);
    printf("Compute GPU forward FFT in-place\n");
#pragma omp target data map(tofrom:xGPU[0:N]) device(devNum)
    {
#pragma omp target variant dispatch use_device_ptr(xGPU) device(devNum)
        {
            statusGPU = DftiComputeForward(descHandleGPU, xGPU);
        }
    }
    // use results now in x and xGPU ...
cleanup:
    DftiFreeDescriptor(&descHandle);
    DftiFreeDescriptor(&descHandleGPU);
    mkl_free(x);
    mkl_free(xGPU);
}

```

For asynchronous execution of multi-call oneMKL computation, the `nowait` clause needs to be used only on the call to the function performing the actual computation (for example, `DftiCompute{Forward,Backward}`). For instance, the following snippet shows how the DFTI example above could be changed to have two, back-to-back, asynchronous (`nowait`) computations dispatched, with a `taskwait` at the end of the second to ensure the completion of both computations before their results are accessed:

```

printf("Compute Intel GPU forward FFT 1 in-place\n");
#pragma omp target data map(tofrom:x1GPU[0:N1], x2GPU[0:N2]) device(devNum)
{
#pragma omp target variant dispatch use_device_ptr(x1GPU) device(devNum) nowait
    {
        status1GPU = DftiComputeForward(descHandle1GPU, x1GPU);
    }
    printf("Compute Intel GPU forward FFT 2 in-place\n");
#pragma omp target variant dispatch use_device_ptr(x2GPU) device(devNum) nowait
    {
        status2GPU = DftiComputeForward(descHandle2GPU, x2GPU);
    }
#pragma omp taskwait
    {
        if (status1GPU != DFTI_NO_ERROR) goto failed;
        if (status2GPU != DFTI_NO_ERROR) goto failed;
    }
}

```

For sparse BLAS computations, the workflow 'create a CSR matrix handle' → 'compute' → 'destroy the CSR matrix handle' must be done so that the offloaded data arrays are alive through the full workflow. For instance, if you are using a target data map, then the workflow must be contained in a single target data region. On the other hand, if the arrays were allocated directly using `omp_target_alloc()` or the Intel Extensions `omp_target_alloc_host/omp_target_alloc_device/omp_target_alloc_shared`, then the workflow must be contained at least in a subset of the scope where those arrays are usable; that is, before the corresponding calls to `omp_target_free`. The following snippet shows how the Sparse BLAS OpenMP Offload example for `mkl_sparse_s_mv()` could be run using a target data map region, where N is the number of rows, M is the number of columns, and NNZ is the number of non-zero entries of the sparse matrix `csrA_gpu`, x is the input vector, and the output is stored in the z array:

```
#pragma omp target data map(to:ia[0:N+1],ja[0:NNZ],a[0:NNZ],x[0:M]) map(tofrom:z[0:N])
device(devNum)
{
    #pragma omp target variant dispatch device(devNum) use_device_ptr(ia, ja, a)
    {
        status_gpu1 = mkl_sparse_s_create_csr(&csrA_gpu, SPARSE_INDEX_BASE_ZERO, N, M, ia,
        ia + 1, ja, a);
    }
    #pragma omp target variant dispatch device(devNum) use_device_ptr(x, z)
    {
        status_gpu2 = mkl_sparse_s_mv(SPARSE_OPERATION_NON_TRANSPOSE, alpha, csrA_gpu,
        descrA, x, beta, z);
    }
    #pragma omp target variant dispatch device(devNum)
    {
        status_gpu3 = mkl_sparse_destroy(csrA_gpu);
    }
}
```

For asynchronous execution of multi-call oneMKL Sparse BLAS computation, the `nowait` clause can be added to the call of the function performing the actual computation (for example, calls to the `mkl_sparse_{s,d}_mv()` function).

As an example, the following snippet shows how the Sparse BLAS example above could be changed to have two asynchronous (`nowait`) computations using the same matrix handle, `csrA_gpu`, but unrelated vector data so there is no read/write dependency between them. Add a `taskwait` at the end of the second execution to ensure the completion of both computations before the `mkl_sparse_destroy()` function is called:

```
#pragma omp target data map(to:ia[0:N+1],ja[0:NNZ],a[0:NNZ],x[0:M],w[0:M])
map(tofrom:y[0:N],z[0:N]) device(devNum)
{
    #pragma omp target variant dispatch device(devNum) use_device_ptr(ia, ja, a)
    {
        status_gpu1 = mkl_sparse_s_create_csr(&csrA_gpu, SPARSE_INDEX_BASE_ZERO, N, M, ia,
        ia + 1, ja, a);
    }

    #pragma omp target variant dispatch device(devNum) use_device_ptr(x, z) nowait
    {
        status_gpu2 = mkl_sparse_s_mv(SPARSE_OPERATION_NON_TRANSPOSE, alpha, csrA_gpu,
        descrA, x, beta, z);
    }

    #pragma omp target variant dispatch device(devNum) use_device_ptr(w, y) nowait
    {
        status_gpu3 = mkl_sparse_s_mv(SPARSE_OPERATION_NON_TRANSPOSE, alpha, csrA_gpu,
        descrA, w, beta, y);
    }
}
```

```

    }
#pragma omp taskwait

#pragma omp target variant dispatch device(devNum)
{
    status_gpu4 = mkl_sparse_destroy(csrA_gpu);
}
}

```

BLAS and Sparse BLAS Routines

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

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

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

When BLAS or Sparse BLAS routines encounter an error, they call the error reporting routine [xerbla](#).

BLAS Routines

Naming Conventions for BLAS Routines

BLAS routine names have the following structure:

`<character> <name> <mod> [_64]`

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

<i>s</i>	real, single precision
<i>c</i>	complex, single precision
<i>d</i>	real, double precision
<i>z</i>	complex, double precision

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

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

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

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

<i>ge</i>	general matrix
<i>gb</i>	general band matrix
<i>sy</i>	symmetric matrix
<i>sp</i>	symmetric matrix (packed storage)

sb	symmetric band matrix
he	Hermitian matrix
hp	Hermitian matrix (packed storage)
hb	Hermitian band matrix
tr	triangular matrix
tp	triangular matrix (packed storage)
tb	triangular band matrix.

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

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

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

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

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

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

On 64-bit platforms, routines with the `_64` suffix support large data arrays in the LP64 interface library and enable you to mix integer types in one application. For example, when an application is linked with the LP64 interface library, `SGEMM` indexes arrays with the 32-bit integer type, while `SGEMM_64` indexes arrays with the 64-bit integer type. For more interface library details, see "Using the ILP64 Interface vs. LP64 Interface" in the developer guide.

The examples below illustrate how to interpret BLAS routine names:

ddot	<code><d> <dot></code> : real and double precision, vector-vector dot product
cdotc	<code><c> <dot> <c></code> : complex and single precision, vector-vector dot product, conjugated
cdotu	<code><c> <dot> <u></code> : complex and single precision, vector-vector dot product, unconjugated
scasum	<code><sc> <asum></code> : real and single-precision output, complex and single-precision input, sum of magnitudes of vector elements
sgemv	<code><s> <ge> <mv></code> : real and single precision, general matrix, matrix-vector product

`ztrmm` `<z>` `<tr>` `<mm>` `_64`: complex and double precision, triangular matrix, matrix-matrix product, 64-bit integer type

Sparse BLAS level 1 naming conventions are similar to those of BLAS level 1. For more information, see [Naming Conventions](#).

C Interface Conventions for BLAS Routines

CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS), provides a C language interface to BLAS routines for Intel® oneAPI Math Kernel Library (oneMKL). While you can call the Fortran implementation of BLAS, for coding in C the CBLAS interface has some advantages such as allowing you to specify column-major or row-major ordering with the `layout` parameter.

For more information about calling Fortran routines from C in general, and specifically about calling BLAS and CBLAS routines, see "Mixed-language Programming with the Intel® oneAPI Math Kernel Library" in the *Intel® oneAPI Math Kernel Library Developer Guide*.

NOTE

This reference contains syntax in C for both the CBLAS interface and the Fortran BLAS routines.

In CBLAS, the Fortran routine names are prefixed with `cblas_` (for example, `dasum` becomes `cblas_dasum`). Names of all CBLAS functions are in lowercase letters. Like BLAS routines, Intel® oneAPI Math Kernel Library provides CBLAS routines with the `_64` suffix (for example, `cblas_dasum_64`) to support large data arrays in the LP64 interface library on 64-bit platforms. For more interface library details, see "Using the ILP64 Interface vs. LP64 Interface" in the developer guide.

Complex functions `?dotc` and `?dotu` become CBLAS subroutines (void functions); they return the complex result via a void pointer, added as the last parameter. CBLAS names of these functions are suffixed with `_sub`. For example, the BLAS function `cdotc` corresponds to `cblas_cdotc_sub`.

WARNING

Users of the CBLAS interface should be aware that the CBLAS are just a C interface to the BLAS, which is based on the FORTRAN standard and subject to the FORTRAN standard restrictions. In particular, the output parameters should not be referenced through more than one argument.

NOTE

This interface is not implemented in the Sparse BLAS Level 2 and Level 3 routines.

The arguments of CBLAS functions comply with the following rules:

- Input arguments are declared with the `const` modifier.
- Non-complex scalar input arguments are passed by value.
- Complex scalar input arguments are passed as void pointers.
- Array arguments are passed by address.
- BLAS character arguments are replaced by the appropriate enumerated type.
- Level 2 and Level 3 routines acquire an additional parameter of type `CBLAS_LAYOUT` as their first argument. This parameter specifies whether two-dimensional arrays are row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

Enumerated Types

The CBLAS interface uses the following enumerated types:

```
enum CBLAS_LAYOUT {
    CblasRowMajor=101, /* row-major arrays */
    CblasColMajor=102}; /* column-major arrays */
```

```
enum CBLAS_TRANSPOSE {
    CblasNoTrans=111,      /* trans='N' */
    CblasTrans=112,        /* trans='T' */
    CblasConjTrans=113};   /* trans='C' */
enum CBLAS_UPLO {
    CblasUpper=121,        /* uplo ='U' */
    CblasLower=122};      /* uplo ='L' */
enum CBLAS_DIAG {
    CblasNonUnit=131,      /* diag ='N' */
    CblasUnit=132};       /* diag ='U' */
enum CBLAS_SIDE {
    CblasLeft=141,         /* side ='L' */
    CblasRight=142};      /* side ='R' */
```

Matrix Storage Schemes for BLAS Routines

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

- *Full storage*: a matrix A is stored in a two-dimensional array a , with the matrix element A_{ij} stored in the array element $a[i + j*lda]$ for column-major layout and $a[j + i*lda]$ for row-major layout, where lda is the leading dimension for the array.
- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly. For column-major layout, the upper or lower triangle of the matrix is packed by columns in a one dimensional array. For row-major layout, the upper or lower triangle of the matrix is packed by rows in a one dimensional array.
- *Band storage*: a band matrix is stored compactly in a two-dimensional array. For column-major layout, columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in a specific row of the array. For row-major layout, rows of the matrix are stored in the corresponding rows of the array, and diagonals of the matrix are stored in a specific column of the array.

For more information on matrix storage schemes, see [Matrix Arguments](#) in the Appendix "Routine and Function Arguments".

Row-Major and Column-Major Layout

The BLAS routines follow the Fortran convention of storing two-dimensional arrays using column-major layout. When calling BLAS routines from C, remember that they require arrays to be in column-major format, not the row-major format that is the convention for C. Unless otherwise specified, the psuedo-code examples for the BLAS routines illustrate matrices stored using column-major layout.

The CBLAS interface allows you to specify either column-major or row-major layout for BLAS Level 2 and Level 3 routines, by setting the *layout* parameter to `CblasColMajor` or `CblasRowMajor`.

BLAS Level 1 Routines and Functions

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

BLAS Level 1 Routine and Function Groups and Their Data Types

Routine or Function Group	Data Types	Description
<code>cblas_?asum</code>	s, d, sc, dz	Sum of vector magnitudes (functions)
<code>cblas_?axpy</code>	s, d, c, z	Scalar-vector product (routines)
<code>cblas_?copy</code>	s, d, c, z	Copy vector (routines)
<code>cblas_?dot</code>	s, d	Dot product (functions)

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

cblas_?asum

Computes the sum of magnitudes of the vector elements.

Syntax

```
float cblas_sasum (const MKL_INT n, const float *x, const MKL_INT incx);
float cblas_scasum (const MKL_INT n, const void *x, const MKL_INT incx);
double cblas_dasum (const MKL_INT n, const double *x, const MKL_INT incx);
double cblas_dzasum (const MKL_INT n, const void *x, const MKL_INT incx);
```

Include Files

- `mkl.h`

Description

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

$$res = |Re x_1| + |Im x_1| + |Re x_2| + |Im x_2| + \dots + |Re x_n| + |Im x_n|,$$

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

where x is a vector with n elements.

Input Parameters

<i>n</i>	Specifies the number of elements in vector <i>x</i> .
<i>x</i>	Array, size at least $(1 + (n-1) * \text{abs}(\text{incx}))$.
<i>incx</i>	Specifies the increment for indexing vector <i>x</i> .

Output Parameters

<i>res</i>	Contains the sum of magnitudes of real and imaginary parts of all elements of the vector.
------------	---

Return Values

Contains the sum of magnitudes of real and imaginary parts of all elements of the vector.

cblas_?axpy

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

Syntax

```
void cblas_saxpy (const MKL_INT n, const float a, const float *x, const MKL_INT incx,
float *y, const MKL_INT incy);

void cblas_daxpy (const MKL_INT n, const double a, const double *x, const MKL_INT incx,
double *y, const MKL_INT incy);

void cblas_caxpy (const MKL_INT n, const void *a, const void *x, const MKL_INT incx,
void *y, const MKL_INT incy);

void cblas_zaxpy (const MKL_INT n, const void *a, const void *x, const MKL_INT incx,
void *y, const MKL_INT incy);
```

Include Files

- mkl.h

Description

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

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

where:

a is a scalar

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

Input Parameters

<i>n</i>	Specifies the number of elements in vectors <i>x</i> and <i>y</i> .
<i>a</i>	Specifies the scalar <i>a</i> .
<i>x</i>	Array, size at least $(1 + (n-1) * \text{abs}(\text{incx}))$.
<i>incx</i>	Specifies the increment for the elements of <i>x</i> .

y Array, size at least $(1 + (n-1) * \text{abs}(incy))$.

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

Output Parameters

y Contains the updated vector *y*.

cblas_?copy

Copies a vector to another vector.

Syntax

```
void cblas_scopy (const MKL_INT n, const float *x, const MKL_INT incx, float *y, const MKL_INT incy);
```

```
void cblas_dcopy (const MKL_INT n, const double *x, const MKL_INT incx, double *y, const MKL_INT incy);
```

```
void cblas_ccopy (const MKL_INT n, const void *x, const MKL_INT incx, void *y, const MKL_INT incy);
```

```
void cblas_zcopy (const MKL_INT n, const void *x, const MKL_INT incx, void *y, const MKL_INT incy);
```

Include Files

- `mk1.h`

Description

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

```
y = x,
```

where *x* and *y* are vectors.

Input Parameters

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

x Array, size at least $(1 + (n-1) * \text{abs}(incx))$.

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

y Array, size at least $(1 + (n-1) * \text{abs}(incy))$.

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

Output Parameters

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

cblas_?copy_batch

Computes a group of vector copies.

Syntax

```
void cblas_scopy_batch (const MKL_INT *n_array, const float **x_array, const MKL_INT
*incx_array, float **y_array, const MKL_INT *incy_array, const MKL_INT group_count,
const MKL_INT *group_size_array);
```

```
void cblas_dcopy_batch (const MKL_INT *n_array, const double **x_array, const MKL_INT
*incx_array, double **y_array, const MKL_INT *incy_array, const MKL_INT group_count,
const MKL_INT *group_size_array);
```

```
void cblas_ccopy_batch (const MKL_INT *n_array, const void **x_array, const MKL_INT
*incx_array, void **y_array, const MKL_INT *incy_array, const MKL_INT group_count,
const MKL_INT *group_size_array);
```

```
void cblas_zcopy_batch (const MKL_INT *n_array, const void **x_array, const MKL_INT
*incx_array, void **y_array, const MKL_INT *incy_array, const MKL_INT group_count,
const MKL_INT *group_size_array);
```

Description

The `cblas_?copy_batch` routines perform a series of vector copies. They are similar to their `cblas_?copy` routine counterparts, but the `cblas_?copy_batch` routines perform vector operations with a group of vectors. Each groups contains vectors with the same parameters (size and increment), while those parameters may vary between groups.

The operation is defined as follows:

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

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

Input Parameters

<code>n_array</code>	Array of size <code>group_count</code> . For the group <code>i</code> , <code>n_i = n_array[i]</code> is the number of elements in the vectors <code>x</code> and <code>y</code> .
<code>x_array</code>	Array of size <code>total_batch_count</code> of pointers used to store <code>x</code> vectors. The array allocated for the <code>x</code> vectors of the group <code>i</code> must be of size at least $(1 + (n_i - 1) * \text{abs}(\text{incx}_i))$.
<code>incx_array</code>	Array of size <code>group_count</code> . For the group <code>i</code> , <code>incx_i = incx_array[i]</code> is the increment (or <i>stride</i>) between two consecutive elements of the vector <code>x</code> .
<code>y_array</code>	Array of size <code>total_batch_count</code> of pointers used to store the output vectors <code>y</code> . The array allocated for the <code>y</code> vectors of the group <code>i</code> must be of size at least $(1 + (n_i - 1) * \text{abs}(\text{incy}_i))$.
<code>incy_array</code>	Array of size <code>group_count</code> . For the group <code>i</code> , <code>incy_i = incy_array[i]</code> is the increment (or <i>stride</i>) between two consecutive elements of the vector <code>y</code> .

group_count	Number of groups. Must be at least 0.
group_size_array	Array of size group_count. The element group_size_array[i] is the number of vectors in the group i. Each element in group_size_array must be at least 0.

Output Parameters

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

cblas_?copy_batch_strided

Computes a group of vector copies.

Syntax

```
void cblas_scopy_batch_strided (const MKL_INT n, const float *x, const MKL_INT incx,
const MKL_INT stridex, float *y, const MKL_INT incy, const MKL_INT stridey, const
MKL_INT batch_size);

void cblas_dcopy_batch_strided (const MKL_INT n, const double *x, const MKL_INT incx,
const MKL_INT stridex, double *y, const MKL_INT incy, const MKL_INT stridey, const
MKL_INT batch_size);

void cblas_ccopy_batch_strided (const MKL_INT n, const void *x, const MKL_INT incx,
const MKL_INT stridex, void *y, const MKL_INT incy, const MKL_INT stridey, const
MKL_INT batch_size);

void cblas_zcopy_batch_strided (const MKL_INT n, const void *x, const MKL_INT incx,
const MKL_INT stridex, void *y, const MKL_INT incy, const MKL_INT stridey, const
MKL_INT batch_size);
```

Description

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

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

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

Input Parameters

n	Number of elements in vectors x and y. Must be at least 0.
x	Array containing the input vectors. Must be of size at least (1 + (n-1)*abs(incx)) + (batch_size - 1) * stridex.
incx	Increment between two consecutive elements of a single vector in x.
stridex	Stride between two consecutive vectors in x. Must be at least (1 + (n-1)*abs(incx)).
y	Array holding the output vectors. Must be of size at least (1 + (n-1)*abs(incy)) + (batch_size - 1) * stridey.
incy	Increment between two consecutive elements of a single vector in y.

<code>stridex</code>	Stride between two consecutive <code>y</code> vectors. Must be at least $(1 + (n-1) * \text{abs}(\text{incy}))$.
<code>batch_size</code>	Number of copy computations to perform; also the number of <code>x</code> and <code>y</code> vectors. Must be at least 0.

Output Parameters

<code>y</code>	Array holding the <code>batch_size</code> copied vectors <code>y</code> .
----------------	---

`cblas_?dot`

Computes a vector-vector dot product.

Syntax

```
float cblas_sdot (const MKL_INT n, const float *x, const MKL_INT incx, const float *y,
const MKL_INT incy);
```

```
double cblas_ddot (const MKL_INT n, const double *x, const MKL_INT incx, const double
*y, const MKL_INT incy);
```

Include Files

- `mkl.h`

Description

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

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

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

Input Parameters

<code>n</code>	Specifies the number of elements in vectors <code>x</code> and <code>y</code> .
<code>x</code>	Array, size at least $(1 + (n-1) * \text{abs}(\text{incx}))$.
<code>incx</code>	Specifies the increment for the elements of <code>x</code> .
<code>y</code>	Array, size at least $(1 + (n-1) * \text{abs}(\text{incy}))$.
<code>incy</code>	Specifies the increment for the elements of <code>y</code> .

Return Values

The result of the dot product of `x` and `y`, if `n` is positive. Otherwise, returns 0.

`cblas_?sdot`

Computes a vector-vector dot product with double precision.

Syntax

```
float cblas_sdsdot (const MKL_INT n, const float sb, const float *sx, const MKL_INT incx, const float *sy, const MKL_INT incy);
```

```
double cblas_dsdot (const MKL_INT n, const float *sx, const MKL_INT incx, const float *sy, const MKL_INT incy);
```

Include Files

- mkl.h

Description

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

Input Parameters

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

Output Parameters

<code>res</code>	Contains the result of the dot product of <code>sx</code> and <code>sy</code> (with <code>sb</code> added for <code>sdsdot</code>), if <code>n</code> is positive. Otherwise, <code>res</code> contains <code>sb</code> for <code>sdsdot</code> and 0 for <code>dsdot</code> .
------------------	---

Return Values

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

cblas_?dotc

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

Syntax

```
void cblas_cdotc_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotc);
```

```
void cblas_zdotc_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotc);
```

Include Files

- `mkl.h`

Description

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

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

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

Input Parameters

<code>n</code>	Specifies the number of elements in vectors x and y .
<code>x</code>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	Specifies the increment for the elements of x .
<code>y</code>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
<code>incy</code>	Specifies the increment for the elements of y .

Output Parameters

<code>dotc</code>	Contains the result of the dot product of the conjugated x and unconjugated y , if n is positive. Otherwise, it contains 0.
-------------------	---

`cblas_?dotu`

Computes a complex vector-vector dot product.

Syntax

```
void cblas_cdotu_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotu);
```

```
void cblas_zdotu_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotu);
```

Include Files

- `mkl.h`

Description

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

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

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

NOTE The `_sub` suffix on `cblas_cdotu_sub` and `cblas_zdotu_sub` is to emphasize that these are subroutines rather than functions (the return value is stored into the `dotu` pointer).

Input Parameters

<code>n</code>	Specifies the number of elements in vectors x and y .
<code>x</code>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<code>incx</code>	Specifies the increment for the elements of x .
<code>y</code>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
<code>incy</code>	Specifies the increment for the elements of y .

Output Parameters

<code>dotu</code>	Contains the result of the dot product of x and y , if n is positive. Otherwise, it contains 0.
-------------------	---

`cblas_?nrm2`

Computes the Euclidean norm of a vector.

Syntax

```
float cblas_snrm2 (const MKL_INT n, const float *x, const MKL_INT incx);
double cblas_dnrm2 (const MKL_INT n, const double *x, const MKL_INT incx);
float cblas_scnrm2 (const MKL_INT n, const void *x, const MKL_INT incx);
double cblas_dznrm2 (const MKL_INT n, const void *x, const MKL_INT incx);
```

Include Files

- `mkl.h`

Description

The `?nrm2` routines perform a vector reduction operation defined as

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

where:

x is a vector,

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

Input Parameters

<i>n</i>	Specifies the number of elements in vector <i>x</i> .
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<i>incx</i>	Specifies the increment for the elements of <i>x</i> .

Return Values

The Euclidean norm of the vector *x*.

cblas_?rot

Performs rotation of points in the plane.

Syntax

```
void cblas_srot (const MKL_INT n, float *x, const MKL_INT incx, float *y, const MKL_INT incy,
const float c, const float s);
void cblas_drot (const MKL_INT n, double *x, const MKL_INT incx, double *y, const MKL_INT incy,
const double c, const double s);
void cblas_crot (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT incy,
const float c, const void* s);
void cblas_zrot (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT incy,
const double c, const void* s);
void cblas_csrot (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT incy,
const float c, const float s);
void cblas_zdrot (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT incy,
const double c, const double s);
```

Description

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

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

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

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

Input Parameters

<i>n</i>	Specifies the number of elements in vectors <i>x</i> and <i>y</i> .
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<i>incx</i>	Specifies the increment for the elements of <i>x</i> .
<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
<i>incy</i>	Specifies the increment for the elements of <i>y</i> .
<i>c</i>	A scalar.
<i>s</i>	A scalar.

Output Parameters

x	Each element is replaced by $c*x + s*y$.
y	Each element is replaced by $c*y - s*x$, or by $c*y - \text{conj}(s)*x$ if s is a complex type.

cblas_?rotg

Computes the parameters for a Givens rotation.

Syntax

```
void cblas_srotg (float *a, float *b, float *c, float *s);
void cblas_drotg (double *a, double *b, double *c, double *s);
void cblas_crotg (void *a, const void *b, float *c, void *s);
void cblas_zrotg (void *a, const void *b, double *c, void *s);
```

Include Files

- mkl.h

Description

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

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

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

Input Parameters

a	Provides the x-coordinate of the point p .
b	Provides the y-coordinate of the point p .

Output Parameters

a	Contains the parameter r associated with the Givens rotation.
b	Contains the parameter z associated with the Givens rotation.

<i>c</i>	Contains the parameter <i>c</i> associated with the Givens rotation.
<i>s</i>	Contains the parameter <i>s</i> associated with the Givens rotation.

cbblas_?rotm

Performs modified Givens rotation of points in the plane.

Syntax

```
void cbblas_srotm (const MKL_INT n, float *x, const MKL_INT incx, float *y, const
MKL_INT incy, const float *param);

void cbblas_drotm (const MKL_INT n, double *x, const MKL_INT incx, double *y, const
MKL_INT incy, const double *param);
```

Include Files

- `mk1.h`

Description

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

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

for $i=1$ to n , where H is a modified Givens transformation matrix whose values are stored in the `param[1]` through `param[4]` array. See discussion on the `param` argument.

Input Parameters

<i>n</i>	Specifies the number of elements in vectors <i>x</i> and <i>y</i> .
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.
<i>incx</i>	Specifies the increment for the elements of <i>x</i> .
<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.
<i>incy</i>	Specifies the increment for the elements of <i>y</i> .
<i>param</i>	Array, size 5. The elements of the <i>param</i> array are: <i>param</i> [0] contains a switch, <i>flag</i> . <i>param</i> [1-4] contain h_{11} , h_{21} , h_{12} , and h_{22} , respectively, the components of the array H .

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

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

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

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

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

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

Output Parameters

<i>x</i>	Each element <i>x</i> [<i>i</i>] is replaced by $h_{11} * x[i] + h_{12} * y[i]$.
<i>y</i>	Each element <i>y</i> [<i>i</i>] is replaced by $h_{21} * x[i] + h_{22} * y[i]$.

cblas_?rotmg

Computes the parameters for a modified Givens rotation.

Syntax

```
void cblas_srotmg (float *d1, float *d2, float *x1, const float y1, float *param);
void cblas_drotmg (double *d1, double *d2, double *x1, const double y1, double *param);
```

Include Files

- mkl.h

Description

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

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

Input Parameters

<i>d1</i>	Provides the scaling factor for the <i>x</i> -coordinate of the input vector.
<i>d2</i>	Provides the scaling factor for the <i>y</i> -coordinate of the input vector.
<i>x1</i>	Provides the <i>x</i> -coordinate of the input vector.
<i>y1</i>	Provides the <i>y</i> -coordinate of the input vector.

Output Parameters

<i>d1</i>	Provides the first diagonal element of the updated matrix.
<i>d2</i>	Provides the second diagonal element of the updated matrix.
<i>x1</i>	Provides the <i>x</i> -coordinate of the rotated vector before scaling.
<i>param</i>	Array, size 5.

The elements of the *param* array are:

param[0] contains a switch, *flag*. the other array elements *param*[1-4] contain the components of the array *H*: h_{11} , h_{21} , h_{12} , and h_{22} , respectively.

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

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

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

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

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

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

cblas_?scal

Computes the product of a vector by a scalar.

Syntax

```
void cblas_sscal (const MKL_INT n, const float a, float *x, const MKL_INT incx);
void cblas_dscal (const MKL_INT n, const double a, double *x, const MKL_INT incx);
void cblas_cscal (const MKL_INT n, const void *a, void *x, const MKL_INT incx);
void cblas_zscal (const MKL_INT n, const void *a, void *x, const MKL_INT incx);
void cblas_csscal (const MKL_INT n, const float a, void *x, const MKL_INT incx);
void cblas_zdscal (const MKL_INT n, const double a, void *x, const MKL_INT incx);
```

Include Files

- mkl.h

Description

The ?scal routines perform a vector operation defined as

$$x = a * x$$

where:

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

Input Parameters

<i>n</i>	Specifies the number of elements in vector <i>x</i> .
<i>a</i>	Specifies the scalar <i>a</i> .

x Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.

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

Output Parameters

x Updated vector *x*.

cblas_?swap

Swaps a vector with another vector.

Syntax

```
void cblas_sswap (const MKL_INT n, float *x, const MKL_INT incx, float *y, const
MKL_INT incy);

void cblas_dswap (const MKL_INT n, double *x, const MKL_INT incx, double *y, const
MKL_INT incy);

void cblas_cswap (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT
incy);

void cblas_zswap (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT
incy);
```

Include Files

- mkl.h

Description

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

Input Parameters

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

x Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.

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

y Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$.

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

Output Parameters

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

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

cblas_i?amax

Finds the index of the element with maximum absolute value.

Syntax

```
CBLAS_INDEX cblas_isamax (const MKL_INT n, const float *x, const MKL_INT incx);
```

```
CBLAS_INDEX cblas_idamax (const MKL_INT n, const double *x, const MKL_INT incx);
CBLAS_INDEX cblas_icamax (const MKL_INT n, const void *x, const MKL_INT incx);
CBLAS_INDEX cblas_izamax (const MKL_INT n, const void *x, const MKL_INT incx);
```

Include Files

- mkl.h

Description

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

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

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

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

Input Parameters

n	Specifies the number of elements in vector x .
x	Array, size at least $(1 + (n-1) * \operatorname{abs}(incx))$.
$incx$	Specifies the increment for the elements of x .

Return Values

Returns the position of vector element that has the largest absolute value such that $x[index-1]$ has the largest absolute value. The index returned is zero-based.

cblas_i?amin

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

Syntax

```
CBLAS_INDEX cblas_isamin (const MKL_INT n, const float *x, const MKL_INT incx);
CBLAS_INDEX cblas_idamin (const MKL_INT n, const double *x, const MKL_INT incx);
CBLAS_INDEX cblas_icamin (const MKL_INT n, const void *x, const MKL_INT incx);
CBLAS_INDEX cblas_izamin (const MKL_INT n, const void *x, const MKL_INT incx);
```

Include Files

- mkl.h

Description

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

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

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

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

Input Parameters

n	On entry, n specifies the number of elements in vector x .
x	Array, size at least $(1+(n-1)*abs(incx))$.
$incx$	Specifies the increment for the elements of x .

Return Values

Indicates the position of vector element with the smallest absolute value such that $x[index-1]$ has the smallest absolute value. The index returned is zero-based.

cblas_?cabs1

Computes absolute value of complex number.

Syntax

```
float cblas_scabs1 (const void *z);
double cblas_dcabs1 (const void *z);
```

Include Files

- mkl.h

Description

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

$$res = |\operatorname{Re}(z)| + |\operatorname{Im}(z)|,$$

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

Input Parameters

z	Scalar.
-----	---------

Return Values

The absolute value of a complex number z .

BLAS Level 2 Routines

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

BLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
<code>cblas_?gbmv</code>	s, d, c, z	Matrix-vector product using a general band matrix
<code>cblas_?gemv</code>	s, d, c, z	Matrix-vector product using a general matrix
<code>cblas_?ger</code>	s, d	Rank-1 update of a general matrix

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

[cblas_?gbmv](#)

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

Syntax

```
void cblas_sgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const float alpha, const float *a, const MKL_INT lda, const float *x, const MKL_INT incx, const float beta, float *y, const MKL_INT incy);
```

```
void cblas_dgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT
m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const double alpha, const
double *a, const MKL_INT lda, const double *x, const MKL_INT incx, const double beta,
double *y, const MKL_INT incy);
```

```
void cblas_cgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT
m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const void *alpha, const void
*a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y,
const MKL_INT incy);
```

```
void cblas_zgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT
m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const void *alpha, const void
*a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y,
const MKL_INT incy);
```

Include Files

- mkl.h

Description

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

$$y := \alpha A x + \beta y,$$

or

$$y := \alpha A' x + \beta y,$$

or

$$y := \alpha \text{conjg}(A') x + \beta y,$$

where:

α and β are scalars,

x and y are vectors,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>trans</i>	Specifies the operation: If $trans = \text{CblasNoTrans}$, then $y := \alpha A x + \beta y$ If $trans = \text{CblasTrans}$, then $y := \alpha A' x + \beta y$ If $trans = \text{CblasConjTrans}$, then $y := \alpha \text{conjg}(A') x + \beta y$
<i>m</i>	Specifies the number of rows of the matrix A . The value of m must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix A . The value of n must be at least zero.
<i>kl</i>	Specifies the number of sub-diagonals of the matrix A .

The value of kl must satisfy $0 \leq kl$.

ku

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

The value of ku must satisfy $0 \leq ku$.

$alpha$

Specifies the scalar $alpha$.

a

Array, size $lda * n$.

Layout = CblasColMajor: Before entry, the leading $(kl + ku + 1)$ by n part of the array a must contain the matrix of coefficients. This matrix must be supplied column-by-column, with the leading diagonal of the matrix in row (ku) of the array, the first super-diagonal starting at position 1 in row $(ku - 1)$, the first sub-diagonal starting at position 0 in row $(ku + 1)$, and so on. Elements in the array a that do not correspond to elements in the band matrix (such as the top left ku by ku triangle) are not referenced.

The following program segment transfers a band matrix from conventional full matrix storage ($matrix$, with leading dimension ldm) to band storage (a , with leading dimension lda):

```
for (j = 0; j < n; j++) {
    k = ku - j;
    for (i = max(0, j-ku); i < min(m, j+kl+1); i++) {
        a[(k+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Layout = CblasRowMajor: Before entry, the leading $(kl + ku + 1)$ by m part of the array a must contain the matrix of coefficients. This matrix must be supplied row-by-row, with the leading diagonal of the matrix in column (kl) of the array, the first super-diagonal starting at position 0 in column $(kl + 1)$, the first sub-diagonal starting at position 1 in row $(kl - 1)$, and so on. Elements in the array a that do not correspond to elements in the band matrix (such as the top left kl by kl triangle) are not referenced.

The following program segment transfers a band matrix from row-major full matrix storage ($matrix$, with leading dimension ldm) to band storage (a , with leading dimension lda):

```
for (i = 0; i < m; i++) {
    k = kl - i;
    for (j = max(0, i-kl); j < min(n, i+ku+1); j++) {
        a[(k+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

lda

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

x

Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$ when $\text{trans} = \text{CblasNoTrans}$, and at least $(1 + (m - 1) * \text{abs}(\text{incx}))$ otherwise. Before entry, the array x must contain the vector x .

$incx$

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

$beta$

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

<i>y</i>	Array, size at least $(1 + (m - 1) * \text{abs}(\text{incy}))$ when <i>trans</i> =CblasNoTrans and at least $(1 + (n - 1) * \text{abs}(\text{incy}))$ otherwise. Before entry, the incremented array <i>y</i> must contain the vector <i>y</i> .
<i>incy</i>	Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Buffer holding the updated vector <i>y</i> .
----------	--

cblas_?gemv

Computes a matrix-vector product using a general matrix.

Syntax

```
void cblas_sgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT
m, const MKL_INT n, const float alpha, const float *a, const MKL_INT lda, const float
*x, const MKL_INT incx, const float beta, float *y, const MKL_INT incy);

void cblas_dgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT
m, const MKL_INT n, const double alpha, const double *a, const MKL_INT lda, const
double *x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);

void cblas_cgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT
m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *x,
const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);

void cblas_zgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT
m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *x,
const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);
```

Include Files

- mkl.h

Description

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

$$y := \alpha A x + \beta y,$$

or

$$y := \alpha A' x + \beta y,$$

or

$$y := \alpha \text{conjg}(A') x + \beta y,$$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>trans</i>	Specifies the operation: if <i>trans</i> =CblasNoTrans, then $y := \alpha * A * x + \beta * y$; if <i>trans</i> =CblasTrans, then $y := \alpha * A' * x + \beta * y$; if <i>trans</i> =CblasConjTrans, then $y := \alpha * \text{conjg}(A') * x + \beta * y$.
<i>m</i>	Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>a</i>	Array, size $lda * k$. For <i>Layout</i> = CblasColMajor, <i>k</i> is <i>n</i> . Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> . For <i>Layout</i> = CblasRowMajor, <i>k</i> is <i>m</i> . Before entry, the leading <i>n</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>lda</i>	Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. For <i>Layout</i> = CblasColMajor, the value of <i>lda</i> must be at least $\max(1, m)$. For <i>Layout</i> = CblasRowMajor, the value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$ when <i>trans</i> =CblasNoTrans and at least $(1 + (m - 1) * \text{abs}(\text{incx}))$ otherwise. Before entry, the incremented array <i>x</i> must contain the vector <i>x</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <i>y</i> need not be set on input.
<i>y</i>	Array, size at least $(1 + (m - 1) * \text{abs}(\text{incy}))$ when <i>trans</i> =CblasNoTrans and at least $(1 + (n - 1) * \text{abs}(\text{incy}))$ otherwise. Before entry with non-zero <i>beta</i> , the incremented array <i>y</i> must contain the vector <i>y</i> .
<i>incy</i>	Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Updated vector <i>y</i> .
----------	---------------------------

cblas_?ger

Performs a rank-1 update of a general matrix.

Syntax

```
void cblas_sger (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const
float alpha, const float *x, const MKL_INT incx, const float *y, const MKL_INT incy,
float *a, const MKL_INT lda);
```

```
void cblas_dger (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const
double alpha, const double *x, const MKL_INT incx, const double *y, const MKL_INT incy,
double *a, const MKL_INT lda);
```

Include Files

- mkl.h

Description

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

$$A := \alpha x y^T + A,$$

where:

α is a scalar,

x is an m -element vector,

y is an n -element vector,

A is an m -by- n general matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>m</i>	Specifies the number of rows of the matrix A . The value of m must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix A . The value of n must be at least zero.
<i>alpha</i>	Specifies the scalar α .
<i>x</i>	Array, size at least $(1 + (m - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the m -element vector x .
<i>incx</i>	Specifies the increment for the elements of x . The value of $incx$ must not be zero.
<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array y must contain the n -element vector y .
<i>incy</i>	Specifies the increment for the elements of y . The value of $incy$ must not be zero.
<i>a</i>	Array, size $lda * k$.

For *Layout* = CblasColMajor, *k* is *n*. Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix *A*.

For *Layout* = CblasRowMajor, *k* is *m*. Before entry, the leading *n*-by-*m* part of the array *a* must contain the matrix *A*.

lda

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

For *Layout* = CblasColMajor, the value of *lda* must be at least $\max(1, m)$.

For *Layout* = CblasRowMajor, the value of *lda* must be at least $\max(1, n)$.

Output Parameters

a

Overwritten by the updated matrix.

cblas_?gerc

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

Syntax

```
void cblas_cgerc (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const
void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void
*a, const MKL_INT lda);
```

```
void cblas_zgerc (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const
void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void
*a, const MKL_INT lda);
```

Include Files

- mkl.h

Description

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

$$A := \alpha x \text{conjg}(y') + A,$$

where:

alpha is a scalar,

x is an *m*-element vector,

y is an *n*-element vector,

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

Input Parameters

Layout

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

m

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

	The value of m must be at least zero.
n	Specifies the number of columns of the matrix A . The value of n must be at least zero.
α	Specifies the scalar α .
x	Array, size at least $(1 + (m - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the m -element vector x .
incx	Specifies the increment for the elements of x . The value of incx must not be zero.
y	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array y must contain the n -element vector y .
incy	Specifies the increment for the elements of y . The value of incy must not be zero.
a	Array, size $\text{lda} * k$. For $\text{Layout} = \text{CblasColMajor}$, k is n . Before entry, the leading m -by- n part of the array a must contain the matrix A . For $\text{Layout} = \text{CblasRowMajor}$, k is m . Before entry, the leading n -by- m part of the array a must contain the matrix A .
lda	Specifies the leading dimension of a as declared in the calling (sub)program. For $\text{Layout} = \text{CblasColMajor}$, the value of lda must be at least $\max(1, m)$. For $\text{Layout} = \text{CblasRowMajor}$, the value of lda must be at least $\max(1, n)$.

Output Parameters

a	Overwritten by the updated matrix.
-----	------------------------------------

cblas_?geru

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

Syntax

```
void cblas_cgeru (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const
void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void
*a, const MKL_INT lda);

void cblas_zgeru (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const
void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void
*a, const MKL_INT lda);
```

Include Files

- mkl.h

Description

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

$$A := \alpha * x * y^T + A,$$

where:

α is a scalar,

x is an m -element vector,

y is an n -element vector,

A is an m -by- n matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>m</i>	Specifies the number of rows of the matrix A . The value of m must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix A . The value of n must be at least zero.
<i>alpha</i>	Specifies the scalar α .
<i>x</i>	Array, size at least $(1 + (m - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the m -element vector x .
<i>incx</i>	Specifies the increment for the elements of x . The value of $incx$ must not be zero.
<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array y must contain the n -element vector y .
<i>incy</i>	Specifies the increment for the elements of y . The value of $incy$ must not be zero.
<i>a</i>	Array, size $lda * k$. For <i>Layout</i> = CblasColMajor, k is n . Before entry, the leading m -by- n part of the array a must contain the matrix A . For <i>Layout</i> = CblasRowMajor, k is m . Before entry, the leading n -by- m part of the array a must contain the matrix A .
<i>lda</i>	Specifies the leading dimension of a as declared in the calling (sub)program. For <i>Layout</i> = CblasColMajor, the value of <i>lda</i> must be at least $\max(1, m)$. For <i>Layout</i> = CblasRowMajor, the value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

a

Overwritten by the updated matrix.

cblas_?hbm

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

Syntax

```
void cblas_chbm (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *x,
const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);
```

```
void cblas_zhbm (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *x,
const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);
```

Include Files

- mkl.h

Description

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

where:

alpha and *beta* are scalars,

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

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the Hermitian band matrix <i>A</i> is used: If <i>uplo</i> = CblasUpper, then the upper triangular part of the matrix <i>A</i> is used. If <i>uplo</i> = CblasLower, then the low triangular part of the matrix <i>A</i> is used.
<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	For <i>uplo</i> = CblasUpper: Specifies the number of super-diagonals of the matrix <i>A</i> . For <i>uplo</i> = CblasLower: Specifies the number of sub-diagonals of the matrix <i>A</i> . The value of <i>k</i> must satisfy $0 \leq k$.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>a</i>	Array, size <i>lda</i> * <i>n</i> .

Layout = CblasColMajor:

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

The following program segment transfers the upper triangular part of a Hermitian band matrix from conventional full matrix storage ($matrix$, with leading dimension ldm) to band storage (a , with leading dimension lda):

```
for (j = 0; j < n; j++) {
    m = k - j;
    for (i = max( 0, j - k); i <= j; i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Before entry with `uplo = CblasLower`, the leading $(k + 1)$ by n part of the array a must contain the lower triangular band part of the Hermitian matrix, supplied column-by-column, with the leading diagonal of the matrix in row 0 of the array, the first sub-diagonal starting at position 0 in row 1, and so on. The bottom right k by k triangle of the array a is not referenced.

The following program segment transfers the lower triangular part of a Hermitian band matrix from conventional full matrix storage ($matrix$, with leading dimension ldm) to band storage (a , with leading dimension lda):

```
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Layout = CblasRowMajor:

Before entry with `uplo = CblasUpper`, the leading $(k + 1)$ -by- n part of array a must contain the upper triangular band part of the Hermitian matrix. The matrix must be supplied row-by-row, with the leading diagonal of the matrix in column 0 of the array, the first super-diagonal starting at position 0 in column 1, and so on. The bottom right k -by- k triangle of array a is not referenced.

The following program segment transfers the upper triangular part of a Hermitian band matrix from row-major full matrix storage ($matrix$ with leading dimension ldm) to row-major band storage (a , with leading dimension lda):

```
for (i = 0; i < n; i++) {
    m = -i;
    for (j = i; j < MIN(n, i+k+1); j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

Before entry with `uplo = CblasLower`, the leading $(k + 1)$ -by- n part of array `a` must contain the lower triangular band part of the Hermitian matrix, supplied row-by-row, with the leading diagonal of the matrix in column k of the array, the first sub-diagonal starting at position 1 in column $k-1$, and so on. The top left k -by- k triangle of array `a` is not referenced.

The following program segment transfers the lower triangular part of a Hermitian row-major band matrix from row-major full matrix storage (`matrix`, with leading dimension `ldm`) to row-major band storage (`a`, with leading dimension `lda`):

```
for (i = 0; i < n; i++) {
    m = k - i;
    for (j = max(0, i-k); j <= i; j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

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

<code>lda</code>	Specifies the leading dimension of <code>a</code> as declared in the calling (sub)program. The value of <code>lda</code> must be at least $(k + 1)$.
<code>x</code>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <code>x</code> must contain the vector <code>x</code> .
<code>incx</code>	Specifies the increment for the elements of <code>x</code> . The value of <code>incx</code> must not be zero.
<code>beta</code>	Specifies the scalar <code>beta</code> .
<code>y</code>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <code>y</code> must contain the vector <code>y</code> .
<code>incy</code>	Specifies the increment for the elements of <code>y</code> . The value of <code>incy</code> must not be zero.

Output Parameters

<code>y</code>	Overwritten by the updated vector <code>y</code> .
----------------	--

`cblas_?hemv`

Computes a matrix-vector product using a Hermitian matrix.

Syntax

```
void cblas_chemv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx,
const void *beta, void *y, const MKL_INT incy);
```

```
void cblas_zhemv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx,
const void *beta, void *y, const MKL_INT incy);
```

Include Files

- `mk1.h`

Description

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

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

where:

alpha and *beta* are scalars,

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

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = CblasUpper, then the upper triangular of the array <i>a</i> is used. If <i>uplo</i> = CblasLower, then the low triangular of the array <i>a</i> is used.
<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>a</i>	Array, size <i>lda</i> * <i>n</i> . Before entry with <i>uplo</i> = CblasUpper, the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = CblasLower, the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>a</i> is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	Specifies the scalar <i>beta</i> . When <i>beta</i> is supplied as zero then <i>y</i> need not be set on input.
<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	Specifies the increment for the elements of <i>y</i> .

The value of *incx* must not be zero.

Output Parameters

y Overwritten by the updated vector *y*.

cblas_?her

Performs a rank-1 update of a Hermitian matrix.

Syntax

```
void cblas_cher (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const void *x, const MKL_INT incx, void *a, const MKL_INT lda);
void cblas_zher (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const void *x, const MKL_INT incx, void *a, const MKL_INT lda);
```

Include Files

- mkl.h

Description

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

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

where:

alpha is a real scalar,

x is an *n*-element vector,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = CblasUpper, then the upper triangular of the array <i>a</i> is used. If <i>uplo</i> = CblasLower, then the low triangular of the array <i>a</i> is used.
<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>x</i>	Array, dimension at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>a</i>	Array, size <i>lda</i> * <i>n</i> .

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

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

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

`lda`

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

Output Parameters

`a`

With `uplo = CblasUpper`, the upper triangular part of the array a is overwritten by the upper triangular part of the updated matrix.

With `uplo = CblasLower`, the lower triangular part of the array a is overwritten by the lower triangular part of the updated matrix.

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

`cblas_?her2`

Performs a rank-2 update of a Hermitian matrix.

Syntax

```
void cblas_cher2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT
incy, void *a, const MKL_INT lda);
```

```
void cblas_zher2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT
incy, void *a, const MKL_INT lda);
```

Include Files

- `mk1.h`

Description

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

$$A := \alpha * x * \text{conjg}(y') + \text{conjg}(\alpha) * y * \text{conjg}(x') + A,$$

where:

α is scalar,

x and y are n -element vectors,

A is an n -by- n Hermitian matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = CblasUpper, then the upper triangular of the array <i>a</i> is used. If <i>uplo</i> = CblasLower, then the low triangular of the array <i>a</i> is used.
<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	Array, size <i>lda</i> * <i>n</i> . Before entry with <i>uplo</i> = CblasUpper, the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = CblasLower, the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>a</i> is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	With <i>uplo</i> = CblasUpper, the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = CblasLower, the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated matrix. If <i>alpha</i> is zero, matrix <i>A</i> is unchanged; otherwise, the imaginary parts of the diagonal elements are set to zero.
----------	---

cblas_?hpmv

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

Syntax

```
void cblas_chpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *ap, const void *x, const MKL_INT incx, const void *beta,
void *y, const MKL_INT incy);
```

```
void cblas_zhpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *ap, const void *x, const MKL_INT incx, const void *beta,
void *y, const MKL_INT incy);
```

Include Files

- mkl.h

Description

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

$$y := \alpha A x + \beta y,$$

where:

α and β are scalars,

x and y are n -element vectors,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	<p>Specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array ap.</p> <p>If $uplo = \text{CblasUpper}$, then the upper triangular part of the matrix A is supplied in the packed array ap.</p> <p>If $uplo = \text{CblasLower}$, then the low triangular part of the matrix A is supplied in the packed array ap.</p>
<i>n</i>	Specifies the order of the matrix A . The value of n must be at least zero.
<i>alpha</i>	Specifies the scalar α .
<i>ap</i>	<p>Array, size at least $((n * (n + 1)) / 2)$.</p> <p>For $Layout = \text{CblasColMajor}$:</p> <p>Before entry with $uplo = \text{CblasUpper}$, the array ap must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that $ap[0]$ contains $A_{1,1}$, $ap[1]$ and $ap[2]$ contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with $uplo = \text{CblasLower}$, the array ap must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that $ap[0]$ contains $A_{1,1}$, $ap[1]$ and $ap[2]$ contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.</p> <p>For $Layout = \text{CblasRowMajor}$:</p>

Before entry with `uplo = CblasUpper`, the array `ap` must contain the upper triangular part of the Hermitian matrix packed sequentially, row-by-row, `ap[0]` contains $A_{1,1}$, `ap[1]` and `ap[2]` contain $A_{1,2}$ and $A_{1,3}$ respectively, and so on. Before entry with `uplo = CblasLower`, the array `ap` must contain the lower triangular part of the Hermitian matrix packed sequentially, row-by-row, so that `ap[0]` contains $A_{1,1}$, `ap[1]` and `ap[2]` contain $A_{2,1}$ and $A_{2,2}$ respectively, and so on.

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

<code>x</code>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <code>x</code> must contain the n -element vector <code>x</code> .
<code>incx</code>	Specifies the increment for the elements of <code>x</code> . The value of <code>incx</code> must not be zero.
<code>beta</code>	Specifies the scalar <code>beta</code> . When <code>beta</code> is equal to zero then <code>y</code> need not be set on input.
<code>y</code>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <code>y</code> must contain the n -element vector <code>y</code> .
<code>incy</code>	Specifies the increment for the elements of <code>y</code> . The value of <code>incy</code> must not be zero.

Output Parameters

<code>y</code>	Overwritten by the updated vector <code>y</code> .
----------------	--

cblas_?hpr

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

Syntax

```
void cblas_chpr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const void *x, const MKL_INT incx, void *ap);
```

```
void cblas_zhpr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const void *x, const MKL_INT incx, void *ap);
```

Include Files

- `mk1.h`

Description

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

$$A := \alpha * x * \text{conjg}(x') + A,$$

where:

`alpha` is a real scalar,

`x` is an n -element vector,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	<p>Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = CblasUpper, the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p> <p>If <i>uplo</i> = CblasLower, the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i>.</p>
<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> . <i>incx</i> must not be zero.
<i>ap</i>	<p>Array, size at least $((n * (n + 1)) / 2)$.</p> <p>For <i>Layout</i> = CblasColMajor:</p> <p>Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i>[0] contains $A_{1,1}$, <i>ap</i>[1] and <i>ap</i>[2] contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on.</p> <p>Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i>[0] contains $A_{1,1}$, <i>ap</i>[1] and <i>ap</i>[2] contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.</p> <p>For <i>Layout</i> = CblasRowMajor:</p> <p>Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, row-by-row, so that <i>ap</i>[0] contains $A_{1,1}$, <i>ap</i>[1] and <i>ap</i>[2] contain $A_{1,2}$ and $A_{1,3}$ respectively, and so on.</p> <p>Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, row-by-row, so that <i>ap</i>[0] contains $A_{1,1}$, <i>ap</i>[1] and <i>ap</i>[2] contain $A_{2,1}$ and $A_{2,2}$ respectively, and so on.</p> <p>The imaginary parts of the diagonal elements need not be set and are assumed to be zero.</p>

Output Parameters

<i>ap</i>	<p>With <i>uplo</i> = CblasUpper, overwritten by the upper triangular part of the updated matrix.</p> <p>With <i>uplo</i> = CblasLower, overwritten by the lower triangular part of the updated matrix.</p>
-----------	---

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

cblas_?hpr2

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

Syntax

```
void cblas_chpr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT
incy, void *ap);
```

```
void cblas_zhpr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT
incy, void *ap);
```

Include Files

- mkl.h

Description

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

$$A := \alpha * x * \text{conjg}(y') + \text{conjg}(\alpha) * y * \text{conjg}(x') + A,$$

where:

alpha is a scalar,

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

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = CblasUpper, then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = CblasLower, then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> .
<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>x</i>	Array, dimension at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.

<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>ap</i>	Array, size at least $((n * (n + 1)) / 2)$. For <i>Layout</i> = CblasColMajor: Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on. For <i>Layout</i> = CblasRowMajor: Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, row-by-row, <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{1,2}$ and $A_{1,3}$ respectively, and so on. Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, row-by-row, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{2,1}$ and $A_{2,2}$ respectively, and so on. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

<i>ap</i>	With <i>uplo</i> = CblasUpper, overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = CblasLower, overwritten by the lower triangular part of the updated matrix. If <i>alpha</i> is zero, matrix <i>A</i> is unchanged; otherwise, the imaginary parts of the diagonal elements need are set to zero.
-----------	--

cblas_?sbmv

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

Syntax

```
void cblas_ssbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const MKL_INT k, const float alpha, const float *a, const MKL_INT lda, const float *x,
const MKL_INT incx, const float beta, float *y, const MKL_INT incy);

void cblas_dsbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const MKL_INT k, const double alpha, const double *a, const MKL_INT lda, const double
*x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);
```

Include Files

- `mkl.h`

Description

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

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

where:

α and β are scalars,

x and y are n -element vectors,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the band matrix A is used: if <i>uplo</i> = CblasUpper - upper triangular part; if <i>uplo</i> = CblasLower - low triangular part.
<i>n</i>	Specifies the order of the matrix A . The value of n must be at least zero.
<i>k</i>	Specifies the number of super-diagonals of the matrix A . The value of k must satisfy $0 \leq k$.
<i>alpha</i>	Specifies the scalar α .
<i>a</i>	<p>Array, size $lda * n$. Before entry with <i>uplo</i> = CblasUpper, the leading $(k + 1)$ by n part of the array a must contain the upper triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row k of the array, the first super-diagonal starting at position 1 in row $(k - 1)$, and so on. The top left k by k triangle of the array a is not referenced.</p> <p>The following program segment transfers the upper triangular part of a symmetric band matrix from conventional full matrix storage (<i>matrix</i>, with leading dimension <i>ldm</i>) to band storage (a, with leading dimension <i>lda</i>):</p> <pre>for (j = 0; j < n; j++) { m = k - j; for (i = max(0, j - k); i <= j; i++) { a[(m+i) + j*lda] = matrix[i + j*ldm]; } }</pre> <p>Before entry with <i>uplo</i> = CblasLower, the leading $(k + 1)$ by n part of the array a must contain the lower triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row 0 of the array, the first sub-diagonal starting at position 0 in row 1, and so on. The bottom right k by k triangle of the array a is not referenced.</p>

The following program segment transfers the lower triangular part of a symmetric band matrix from conventional full matrix storage (*matrix*, with leading dimension *ldm*) to band storage (*a*, with leading dimension *lda*):

```
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Layout = CblasRowMajor:

Before entry with *uplo* = CblasUpper, the leading $(k + 1)$ -by-*n* part of array *a* must contain the upper triangular band part of the symmetric matrix. The matrix must be supplied row-by-row, with the leading diagonal of the matrix in column 0 of the array, the first super-diagonal starting at position 0 in column 1, and so on. The bottom right *k*-by-*k* triangle of array *a* is not referenced.

The following program segment transfers the upper triangular part of a symmetric band matrix from row-major full matrix storage (*matrix* with leading dimension *ldm*) to row-major band storage (*a*, with leading dimension *lda*):

```
for (i = 0; i < n; i++) {
    m = -i;
    for (j = i; j < MIN(n, i+k+1); j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

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

The following program segment transfers the lower triangular part of a symmetric row-major band matrix from row-major full matrix storage (*matrix*, with leading dimension *ldm*) to row-major band storage (*a*, with leading dimension *lda*):

```
for (i = 0; i < n; i++) {
    m = k - i;
    for (j = max(0, i-k); j <= i; j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

lda

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

x

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

incx

Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

<i>beta</i>	Specifies the scalar <i>beta</i> .
<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the vector <i>y</i> .
<i>incy</i>	Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

cblas_?spmv

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

Syntax

```
void cblas_sspmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *ap, const float *x, const MKL_INT incx, const float
beta, float *y, const MKL_INT incy);
```

```
void cblas_dspmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *ap, const double *x, const MKL_INT incx, const double
beta, double *y, const MKL_INT incy);
```

Include Files

- `mkl.h`

Description

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

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

where:

alpha and *beta* are scalars,

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

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = CblasUpper, then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = CblasLower, then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> .

<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>ap</i>	<p>Array, size at least $((n * (n + 1)) / 2)$.</p> <p>For <i>Layout</i> = CblasColMajor:</p> <p>Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>[0] contains $A_{1,1}$, <i>ap</i>[1] and <i>ap</i>[2] contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i>[0] contains $A_{1,1}$, <i>ap</i>[1] and <i>ap</i>[2] contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.</p> <p>For <i>Layout</i> = CblasRowMajor:</p> <p>Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, row-by-row, <i>ap</i>[0] contains $A_{1,1}$, <i>ap</i>[1] and <i>ap</i>[2] contain $A_{1,2}$ and $A_{1,3}$ respectively, and so on. Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, row-by-row, so that <i>ap</i>[0] contains $A_{1,1}$, <i>ap</i>[1] and <i>ap</i>[2] contain $A_{2,1}$ and $A_{2,2}$ respectively, and so on.</p>
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	<p>Specifies the increment for the elements of <i>x</i>.</p> <p>The value of <i>incx</i> must not be zero.</p>
<i>beta</i>	<p>Specifies the scalar <i>beta</i>.</p> <p>When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.</p>
<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	<p>Specifies the increment for the elements of <i>y</i>.</p> <p>The value of <i>incy</i> must not be zero.</p>

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

cblas_?spr

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

Syntax

```
void cblas_sspr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *x, const MKL_INT incx, float *ap);
```

```
void cblas_dspr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *x, const MKL_INT incx, double *ap);
```

Include Files

- `mkl.h`

Description

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

$$a := \alpha * x * x' + A,$$

where:

α is a real scalar,

x is an n -element vector,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the matrix A is supplied in the packed array <i>ap</i> . If <i>uplo</i> = CblasUpper, then the upper triangular part of the matrix A is supplied in the packed array <i>ap</i> . If <i>uplo</i> = CblasLower, then the low triangular part of the matrix A is supplied in the packed array <i>ap</i> .
<i>n</i>	Specifies the order of the matrix A . The value of n must be at least zero.
<i>alpha</i>	Specifies the scalar α .
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n -element vector x .
<i>incx</i>	Specifies the increment for the elements of x . The value of <i>incx</i> must not be zero.
<i>ap</i>	For <i>Layout</i> = CblasColMajor: Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on. For <i>Layout</i> = CblasRowMajor: Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, row-by-row, <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{1,2}$ and $A_{1,3}$ respectively, and so on.

Before entry with `uplo = CblasLower`, the array `ap` must contain the lower triangular part of the symmetric matrix packed sequentially, row-by-row, so that `ap[0]` contains $A_{1,1}$, `ap[1]` and `ap[2]` contain $A_{2,1}$ and $A_{2,2}$ respectively, and so on.

Output Parameters

`ap`

With `uplo = CblasUpper`, overwritten by the upper triangular part of the updated matrix.

With `uplo = CblasLower`, overwritten by the lower triangular part of the updated matrix.

cblas_?spr2

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

Syntax

```
void cblas_sspr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *x, const MKL_INT incx, const float *y, const MKL_INT
incy, float *ap);
```

```
void cblas_dspr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *x, const MKL_INT incx, const double *y, const MKL_INT
incy, double *ap);
```

Include Files

- `mkl.h`

Description

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

$$A := \alpha x x' + \alpha y y' + A,$$

where:

α is a scalar,

x and y are n -element vectors,

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

Input Parameters

`Layout`

Specifies whether two-dimensional array storage is row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

`uplo`

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

If `uplo = CblasUpper`, then the upper triangular part of the matrix A is supplied in the packed array `ap`.

If `uplo = CblasLower`, then the low triangular part of the matrix A is supplied in the packed array `ap`.

<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>ap</i>	For <i>Layout</i> = CblasColMajor: Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on. For <i>Layout</i> = CblasRowMajor: Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, row-by-row, <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{1,2}$ and $A_{1,3}$ respectively, and so on. Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, row-by-row, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{2,1}$ and $A_{2,2}$ respectively, and so on.

Output Parameters

<i>ap</i>	With <i>uplo</i> = CblasUpper, overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = CblasLower, overwritten by the lower triangular part of the updated matrix.
-----------	--

cblas_?symv

Computes a matrix-vector product for a symmetric matrix.

Syntax

```
void cblas_ssymv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *a, const MKL_INT lda, const float *x, const MKL_INT
incx, const float beta, float *y, const MKL_INT incy);
```

```
void cblas_dsymv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *a, const MKL_INT lda, const double *x, const MKL_INT
incx, const double beta, double *y, const MKL_INT incy);
```

Include Files

- mkl.h

Description

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

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

where:

alpha and *beta* are scalars,

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

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = CblasUpper, then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = CblasLower, then the low triangular part of the array <i>a</i> is used.
<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>a</i>	Array, size <i>lda</i> * <i>n</i> . Before entry with <i>uplo</i> = CblasUpper, the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix <i>A</i> and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = CblasLower, the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix <i>A</i> and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	Specifies the scalar <i>beta</i> . When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.

<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

cblas_?syr

Performs a rank-1 update of a symmetric matrix.

Syntax

```
void cblas_ssyrr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *x, const MKL_INT incx, float *a, const MKL_INT lda);
void cblas_dsyr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *x, const MKL_INT incx, double *a, const MKL_INT lda);
```

Include Files

- mkl.h

Description

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

$$A := \alpha x x' + A,$$

where:

alpha is a real scalar,

x is an *n*-element vector,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = CblasUpper, then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = CblasLower, then the low triangular part of the array <i>a</i> is used.
<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>x</i>	Array, size at least $(1 + (n-1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> .

The value of *incx* must not be zero.

a

Array, size $lda \times n$.

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

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

lda

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

Output Parameters

a

With *uplo* = CblasUpper, the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = CblasLower, the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.

cblas_?syr2

Performs a rank-2 update of a symmetric matrix.

Syntax

```
void cblas_ssy2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *x, const MKL_INT incx, const float *y, const MKL_INT
incy, float *a, const MKL_INT lda);
```

```
void cblas_dsyr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *x, const MKL_INT incx, const double *y, const MKL_INT
incy, double *a, const MKL_INT lda);
```

Include Files

- mkl.h

Description

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

$$A := \alpha x y' + \alpha y x' + A,$$

where:

alpha is scalar,

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

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

Input Parameters

Layout

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

<i>uplo</i>	Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = CblasUpper, then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = CblasLower, then the low triangular part of the array <i>a</i> is used.
<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incy}))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	Array, size <i>lda</i> * <i>n</i> . Before entry with <i>uplo</i> = CblasUpper, the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = CblasLower, the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	With <i>uplo</i> = CblasUpper, the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = CblasLower, the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated matrix.
----------	--

cblas_?tbmv

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

Syntax

```
void cblas_stbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
float *a, const MKL_INT lda, float *x, const MKL_INT incx);

void cblas_dtbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
double *a, const MKL_INT lda, double *x, const MKL_INT incx);
```



```
void cblas_ctbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
void *a, const MKL_INT lda, void *x, const MKL_INT incx);

void cblas_ztbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
void *a, const MKL_INT lda, void *x, const MKL_INT incx);
```

Include Files

- mkl.h

Description

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

$x := A * x$, or $x := A' * x$, or $x := \text{conjg}(A') * x$,

where:

x is an n -element vector,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the matrix A is an upper or lower triangular matrix: $uplo = \text{CblasUpper}$ if $uplo = \text{CblasLower}$, then the matrix is low triangular.
<i>trans</i>	Specifies the operation: if $trans = \text{CblasNoTrans}$, then $x := A * x$; if $trans = \text{CblasTrans}$, then $x := A' * x$; if $trans = \text{CblasConjTrans}$, then $x := \text{conjg}(A') * x$.
<i>diag</i>	Specifies whether the matrix A is unit triangular: if $diag = \text{CblasUnit}$ then the matrix is unit triangular; if $diag = \text{CblasNonUnit}$, then the matrix is not unit triangular.
<i>n</i>	Specifies the order of the matrix A . The value of n must be at least zero.
<i>k</i>	On entry with $uplo = \text{CblasUpper}$ specifies the number of super-diagonals of the matrix A . On entry with $uplo = \text{CblasLower}$, k specifies the number of sub-diagonals of the matrix a . The value of k must satisfy $0 \leq k$.
<i>a</i>	Array, size $lda * n$. $Layout = \text{CblasColMajor}$:

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

```
for (j = 0; j < n; j++) {
    m = k - j;
    for (i = max( 0, j - k); i <= j; i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Before entry with `uplo = CblasLower`, the leading $(k + 1)$ by n part of the array `a` must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 0 of the array, the first sub-diagonal starting at position 0 in row 1, and so on. The bottom right k by k triangle of the array `a` is not referenced. The following program segment transfers a lower triangular band matrix from conventional full matrix storage (`matrix`, with leading dimension `ldm`) to band storage (`a`, with leading dimension `lda`):

```
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Note that when `diag = CblasUnit`, the elements of the array `a` corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

Layout = CblasRowMajor:

Before entry with `uplo = CblasUpper`, the leading $(k + 1)$ -by- n part of array `a` must contain the upper triangular band part of the matrix of coefficients. The matrix must be supplied row-by-row, with the leading diagonal of the matrix in column 0 of the array, the first super-diagonal starting at position 0 in column 1, and so on. The bottom right k -by- k triangle of array `a` is not referenced.

The following program segment transfers the upper triangular part of a Hermitian band matrix from row-major full matrix storage (`matrix` with leading dimension `ldm`) to row-major band storage (`a`, with leading dimension `lda`):

```
for (i = 0; i < n; i++) {
    m = -i;
    for (j = i; j < MIN(n, i+k+1); j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

Before entry with `uplo = CblasLower`, the leading $(k + 1)$ -by- n part of array `a` must contain the lower triangular band part of the matrix of coefficients, supplied row-by-row, with the leading diagonal of the matrix in column k of the array, the first sub-diagonal starting at position 1 in column $k-1$, and so on. The top left k -by- k triangle of array `a` is not referenced.

The following program segment transfers the lower triangular part of a Hermitian row-major band matrix from row-major full matrix storage (`matrix`, with leading dimension `ldm`) to row-major band storage (`a`, with leading dimension `lda`):

```
for (i = 0; i < n; i++) {
    m = k - i;
    for (j = max(0, i-k); j <= i; j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

`lda`

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

`x`

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

`incx`

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

Output Parameters

`x`

Overwritten with the transformed vector `x`.

`cblas_?tbsv`

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

Syntax

```
void cblas_stbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
float *a, const MKL_INT lda, float *x, const MKL_INT incx);
```

```
void cblas_dtbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
double *a, const MKL_INT lda, double *x, const MKL_INT incx);
```

```
void cblas_ctbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
void *a, const MKL_INT lda, void *x, const MKL_INT incx);
```

```
void cblas_ztbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
void *a, const MKL_INT lda, void *x, const MKL_INT incx);
```

Include Files

- `mkl.h`

Description

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

$A*x = b$, or $A'*x = b$, or $\text{conjg}(A')*x = b$,

where:

b and x are n -element vectors,

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

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

Such tests must be performed before calling this routine.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the matrix A is an upper or lower triangular matrix: if <i>uplo</i> = CblasUpper the matrix is upper triangular; if <i>uplo</i> = CblasLower, the matrix is low triangular.
<i>trans</i>	Specifies the system of equations: if <i>trans</i> =CblasNoTrans, then $A*x = b$; if <i>trans</i> =CblasTrans, then $A'*x = b$; if <i>trans</i> =CblasConjTrans, then $\text{conjg}(A')*x = b$.
<i>diag</i>	Specifies whether the matrix A is unit triangular: if <i>diag</i> = CblasUnit then the matrix is unit triangular; if <i>diag</i> = CblasNonUnit, then the matrix is not unit triangular.
<i>n</i>	Specifies the order of the matrix A . The value of n must be at least zero.
<i>k</i>	On entry with <i>uplo</i> = CblasUpper, k specifies the number of super-diagonals of the matrix A . On entry with <i>uplo</i> = CblasLower, k specifies the number of sub-diagonals of the matrix A . The value of k must satisfy $0 \leq k$.
<i>a</i>	Array, size $lda*n$. <i>Layout</i> = CblasColMajor: Before entry with <i>uplo</i> = CblasUpper, the leading $(k + 1)$ by n part of the array a must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row k of the array, the first super-diagonal starting at position 1 in row $(k - 1)$, and so on. The top left k by k triangle of the array a is not referenced.

The following program segment transfers an upper triangular band matrix from conventional full matrix storage (*matrix*, with leading dimension *ldm*) to band storage (*a*, with leading dimension *lda*):

```
for (j = 0; j < n; j++) {
    m = k - j;
    for (i = max( 0, j - k); i <= j; i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Before entry with *uplo* = CblasLower, the leading $(k + 1)$ by n part of the array *a* must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 0 of the array, the first sub-diagonal starting at position 0 in row 1, and so on. The bottom right k by k triangle of the array *a* is not referenced.

The following program segment transfers a lower triangular band matrix from conventional full matrix storage (*matrix*, with leading dimension *ldm*) to band storage (*a*, with leading dimension *lda*):

```
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

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

Layout = CblasRowMajor:

Before entry with *uplo* = CblasUpper, the leading $(k + 1)$ -by- n part of array *a* must contain the upper triangular band part of the matrix of coefficients. The matrix must be supplied row-by-row, with the leading diagonal of the matrix in column 0 of the array, the first super-diagonal starting at position 0 in column 1, and so on. The bottom right k -by- k triangle of array *a* is not referenced.

The following program segment transfers the upper triangular part of a Hermitian band matrix from row-major full matrix storage (*matrix* with leading dimension *ldm*) to row-major band storage (*a*, with leading dimension *lda*):

```
for (i = 0; i < n; i++) {
    m = -i;
    for (j = i; j < MIN(n, i+k+1); j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

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

The following program segment transfers the lower triangular part of a Hermitian row-major band matrix from row-major full matrix storage (*matrix*, with leading dimension *ldm*) to row-major band storage (*a*, with leading dimension *lda*):

```
for (i = 0; i < n; i++) {
    m = k - i;
    for (j = max(0, i-k); j <= i; j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

lda

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

x

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

incx

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

Output Parameters

x

Overwritten with the solution vector *x*.

cblas_?tpmv

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

Syntax

```
void cblas_stpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *ap, float
*x, const MKL_INT incx);
```

```
void cblas_dtpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *ap, double
*x, const MKL_INT incx);
```

```
void cblas_ctpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void *x,
const MKL_INT incx);
```

```
void cblas_ztpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void *x,
const MKL_INT incx);
```

Include Files

- mkl.h

Description

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

$x := A^*x$, or $x := A'^*x$, or $x := \text{conjg}(A')^*x$,

where:

x is an n -element vector,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the matrix A is upper or lower triangular: $uplo = CblasUpper$ if $uplo = CblasLower$, then the matrix is low triangular.
<i>trans</i>	Specifies the operation: if $trans = CblasNoTrans$, then $x := A * x$; if $trans = CblasTrans$, then $x := A' * x$; if $trans = CblasConjTrans$, then $x := \text{conjg}(A') * x$.
<i>diag</i>	Specifies whether the matrix A is unit triangular: if $diag = CblasUnit$ then the matrix is unit triangular; if $diag = CblasNonUnit$, then the matrix is not unit triangular.
<i>n</i>	Specifies the order of the matrix A . The value of n must be at least zero.
<i>ap</i>	Array, size at least $((n * (n + 1)) / 2)$. For $Layout = CblasColMajor$: Before entry with $uplo = CblasUpper$, the array ap must contain the upper triangular matrix packed sequentially, column-by-column, so that respectively, and so on. Before entry with $uplo = CblasLower$ $ap[0]$ contains $A_{1,1}$, $ap[1]$ and $ap[2]$ contain $A_{1,2}$ and $A_{2,2}$, the array ap must contain the lower triangular matrix packed sequentially, column-by-column, so that $ap[0]$ contains $A_{1,1}$, $ap[1]$ and $ap[2]$ contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on. When $diag = CblasUnit$, the diagonal elements of a are not referenced, but are assumed to be unity. For $Layout = CblasRowMajor$: Before entry with $uplo = CblasUpper$, the array ap must contain the upper triangular matrix packed sequentially, row-by-row, $ap[0]$ contains $A_{1,1}$, $ap[1]$ and $ap[2]$ contain $A_{1,2}$ and $A_{1,3}$ respectively, and so on. Before entry with $uplo = CblasLower$, the array ap must contain the lower triangular matrix packed sequentially, row-by-row, so that $ap[0]$ contains $A_{1,1}$, $ap[1]$ and $ap[2]$ contain $A_{2,1}$ and $A_{2,2}$ respectively, and so on.
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the n -element vector x .
<i>incx</i>	Specifies the increment for the elements of x . The value of $incx$ must not be zero.

Output Parameters

x

Overwritten with the transformed vector x .

cblas_?tpsv

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

Syntax

```
void cblas_stpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *ap, float
*x, const MKL_INT incx);
```

```
void cblas_dtpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *ap, double
*x, const MKL_INT incx);
```

```
void cblas_ctpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void *x,
const MKL_INT incx);
```

```
void cblas_ztpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void *x,
const MKL_INT incx);
```

Include Files

- mkl.h

Description

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

$A*x = b$, or $A'*x = b$, or $\text{conjg}(A')*x = b$,

where:

b and x are n -element vectors,

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

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

Such tests must be performed before calling this routine.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the matrix A is upper or lower triangular: $uplo = \text{CblasUpper}$ if $uplo = \text{CblasLower}$, then the matrix is low triangular.
<i>trans</i>	Specifies the system of equations: if $trans = \text{CblasNoTrans}$, then $A*x = b$; if $trans = \text{CblasTrans}$, then $A'*x = b$;

	if <i>trans</i> =CblasConjTrans, then $\text{conjg}(A') * x = b$.
<i>diag</i>	Specifies whether the matrix <i>A</i> is unit triangular: if <i>diag</i> = CblasUnit then the matrix is unit triangular; if <i>diag</i> = CblasNonUnit , then the matrix is not unit triangular.
<i>n</i>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>ap</i>	Array, size at least $((n * (n + 1)) / 2)$. For <i>Layout</i> = CblasColMajor: Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the triangular matrix packed sequentially, column-by-column, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the triangular matrix packed sequentially, column-by-column, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on. For <i>Layout</i> = CblasRowMajor: Before entry with <i>uplo</i> = CblasUpper, the array <i>ap</i> must contain the upper triangular part of the triangular matrix packed sequentially, row-by-row, <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{1,2}$ and $A_{1,3}$ respectively, and so on. Before entry with <i>uplo</i> = CblasLower, the array <i>ap</i> must contain the lower triangular part of the triangular matrix packed sequentially, row-by-row, so that <i>ap</i> [0] contains $A_{1,1}$, <i>ap</i> [1] and <i>ap</i> [2] contain $A_{2,1}$ and $A_{2,2}$ respectively, and so on. When <i>diag</i> = CblasUnit, the diagonal elements of <i>a</i> are not referenced, but are assumed to be unity.
<i>x</i>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element right-hand side vector <i>b</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten with the solution vector <i>x</i> .
----------	---

cblas_?trmv

Computes a matrix-vector product using a triangular matrix.

Syntax

```
void cblas_strmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *a, const
MKL_INT lda, float *x, const MKL_INT incx);
```

```
void cblas_dtrmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *a, const
MKL_INT lda, double *x, const MKL_INT incx);
```

```
void cblas_ctrmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const
MKL_INT lda, void *x, const MKL_INT incx);
```

```
void cblas_ztrmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const
MKL_INT lda, void *x, const MKL_INT incx);
```

Include Files

- mkl.h

Description

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

$x := A*x$, or $x := A'*x$, or $x := \text{conjg}(A')*x$,

where:

x is an n -element vector,

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

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the matrix A is upper or lower triangular: $uplo = \text{CblasUpper}$ if $uplo = \text{CblasLower}$, then the matrix is low triangular.
<i>trans</i>	Specifies the operation: if $trans = \text{CblasNoTrans}$, then $x := A*x$; if $trans = \text{CblasTrans}$, then $x := A'*x$; if $trans = \text{CblasConjTrans}$, then $x := \text{conjg}(A')*x$.
<i>diag</i>	Specifies whether the matrix A is unit triangular: if $diag = \text{CblasUnit}$ then the matrix is unit triangular; if $diag = \text{CblasNonUnit}$, then the matrix is not unit triangular.
<i>n</i>	Specifies the order of the matrix A . The value of n must be at least zero.
<i>a</i>	Array, size $lda*n$. Before entry with $uplo = \text{CblasUpper}$, the leading n -by- n upper triangular part of the array a must contain the upper triangular matrix and the strictly lower triangular part of a is not referenced. Before entry with $uplo = \text{CblasLower}$, the leading n -by- n lower triangular part of the array a must contain the lower triangular matrix and the strictly upper triangular part of a is not referenced. When $diag = \text{CblasUnit}$, the diagonal elements of a are not referenced either, but are assumed to be unity.
<i>lda</i>	Specifies the leading dimension of a as declared in the calling (sub)program. The value of lda must be at least $\max(1, n)$.

<code>x</code>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <code>x</code> must contain the n -element vector <code>x</code> .
<code>incx</code>	Specifies the increment for the elements of <code>x</code> . The value of <code>incx</code> must not be zero.

Output Parameters

<code>x</code>	Overwritten with the transformed vector <code>x</code> .
----------------	--

`cblas_?trsv`

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

Syntax

```
void cblas_strsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *a, const
MKL_INT lda, float *x, const MKL_INT incx);

void cblas_dtrsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *a, const
MKL_INT lda, double *x, const MKL_INT incx);

void cblas_ctrsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const
MKL_INT lda, void *x, const MKL_INT incx);

void cblas_ztrsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const
MKL_INT lda, void *x, const MKL_INT incx);
```

Include Files

- `mkl.h`

Description

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

$A * x = b$, or $A' * x = b$, or $\text{conjg}(A') * x = b$,

where:

b and x are n -element vectors,

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

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

Such tests must be performed before calling this routine.

Input Parameters

<code>Layout</code>	Specifies whether two-dimensional array storage is row-major (<code>CblasRowMajor</code>) or column-major (<code>CblasColMajor</code>).
<code>uplo</code>	Specifies whether the matrix A is upper or lower triangular:

	<code>uplo = CblasUpper</code> if <code>uplo = CblasLower</code> , then the matrix is low triangular.
<code>trans</code>	Specifies the systems of equations: if <code>trans=CblasNoTrans</code> , then $A*x = b$; if <code>trans=CblasTrans</code> , then $A'*x = b$; if <code>trans=CblasConjTrans</code> , then $\text{oconjg}(A')*x = b$.
<code>diag</code>	Specifies whether the matrix <i>A</i> is unit triangular: if <code>diag = CblasUnit</code> then the matrix is unit triangular; if <code>diag = CblasNonUnit</code> , then the matrix is not unit triangular.
<code>n</code>	Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<code>a</code>	Array, size $lda*n$. Before entry with <code>uplo = CblasUpper</code> , the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <code>uplo = CblasLower</code> , the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced. When <code>diag = CblasUnit</code> , the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.
<code>lda</code>	Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<code>x</code>	Array, size at least $(1 + (n - 1)*\text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element right-hand side vector <i>b</i> .
<code>incx</code>	Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.

Output Parameters

<code>x</code>	Overwritten with the solution vector <i>x</i> .
----------------	---

BLAS Level 3 Routines

BLAS Level 3 routines perform matrix-matrix operations. The following table lists the BLAS Level 3 routine groups and the data types associated with them.

BLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
<code>cblas_?gemm</code>	s, d, c, z	Computes a matrix-matrix product with general matrices.
<code>cblas_?hemm</code>	c, z	Computes a matrix-matrix product where one input matrix is Hermitian.
<code>cblas_?herk</code>	c, z	Performs a Hermitian rank-k update.
<code>cblas_?her2k</code>	c, z	Performs a Hermitian rank-2k update.

Routine Group	Data Types	Description
<code>cblas_?symm</code>	s, d, c, z	Computes a matrix-matrix product where one input matrix is symmetric.
<code>cblas_?syrk</code>	s, d, c, z	Performs a symmetric rank-k update.
<code>cblas_?syr2k</code>	s, d, c, z	Performs a symmetric rank-2k update.
<code>cblas_?trmm</code>	s, d, c, z	Computes a matrix-matrix product where one input matrix is triangular.
<code>cblas_?trsm</code>	s, d, c, z	Solves a triangular matrix equation.

Symmetric Multiprocessing Version of Intel® MKL

Many applications spend considerable time executing BLAS routines. This time can be scaled by the number of processors available on the system through using the symmetric multiprocessing (SMP) feature built into the Intel® oneMKL. The performance enhancements based on the parallel use of the processors are available without any programming effort on your part.

To enhance performance, the library uses the following methods:

- The BLAS functions are blocked where possible to restructure the code in a way that increases the localization of data reference, enhances cache memory use, and reduces the dependency on the memory bus.
- The code is distributed across the processors to maximize parallelism.

Product and Performance Information

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

Notice revision #20201201

`cblas_?gemm`

Computes a matrix-matrix product with general matrices.

Syntax

```
void cblas_hgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const
MKL_F16 alpha, const MKL_F16 *a, const MKL_INT lda, const MKL_F16 *b, const MKL_INT
ldb, const MKL_F16 beta, MKL_F16 *c, const MKL_INT ldc);
```

```
void cblas_sgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const float
alpha, const float *a, const MKL_INT lda, const float *b, const MKL_INT ldb, const
float beta, float *c, const MKL_INT ldc);
```

```
void cblas_dgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const double
alpha, const double *a, const MKL_INT lda, const double *b, const MKL_INT ldb, const
double beta, double *c, const MKL_INT ldc);
```

```
void cblas_cgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void
*alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void
*beta, void *c, const MKL_INT ldc);
```

```
void cblas_zgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void
*alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void
*beta, void *c, const MKL_INT ldc);
```

Include Files

- `mkl.h`

Description

The `?gemm` routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, with general matrices. The operation is defined as

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$$

where:

$\text{op}(X)$ is one of $\text{op}(X) = X$, or $\text{op}(X) = X^T$, or $\text{op}(X) = X^H$,

α and β are scalars,

A , B and C are matrices:

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

See also:

- [?gemm3m](#), BLAS-like extension routines, that use matrix multiplication for similar matrix-matrix operations

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (<code>CblasRowMajor</code>) or column-major (<code>CblasColMajor</code>).
<i>transa</i>	Specifies the form of $\text{op}(A)$ used in the matrix multiplication: <ul style="list-style-type: none"> • if <i>transa</i>=<code>CblasNoTrans</code>, then $\text{op}(A) = A$; • if <i>transa</i>=<code>CblasTrans</code>, then $\text{op}(A) = A^T$; • if <i>transa</i>=<code>CblasConjTrans</code>, then $\text{op}(A) = A^H$.
<i>transb</i>	Specifies the form of $\text{op}(B)$ used in the matrix multiplication: <ul style="list-style-type: none"> • if <i>transb</i>=<code>CblasNoTrans</code>, then $\text{op}(B) = B$; • if <i>transb</i>=<code>CblasTrans</code>, then $\text{op}(B) = B^T$; • if <i>transb</i>=<code>CblasConjTrans</code>, then $\text{op}(B) = B^H$.
<i>m</i>	Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C . The value of <i>m</i> must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C . The value of <i>n</i> must be at least zero.
<i>k</i>	Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of <i>k</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .

a

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans or <i>transa</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size <i>lda</i> * <i>m</i> . Before entry, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>Layout</i> = CblasRowMajor	Array, size <i>lda</i> * <i>m</i> . Before entry, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .

lda

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

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans or <i>transa</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1, m)$.	<i>lda</i> must be at least $\max(1, k)$
<i>Layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1, k)$	<i>lda</i> must be at least $\max(1, m)$.

b

	<i>transb</i> =CblasNoTrans	<i>transb</i> =CblasTrans or <i>transb</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	Array, size <i>ldb</i> by <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>ldb</i> by <i>k</i> . Before entry the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>Layout</i> = CblasRowMajor	Array, size <i>ldb</i> by <i>k</i> . Before entry the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>ldb</i> by <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .

ldb

Specifies the leading dimension of *b* as declared in the calling (sub)program.

When *transb*=CblasNoTrans, then *ldb* must be at least $\max(1, k)$, otherwise *ldb* must be at least $\max(1, n)$.

	<i>transb</i> =CblasNoTrans	<i>transb</i> =CblasTrans or <i>transb</i> =CblasConjTrans
--	-----------------------------	---

<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> must be at least $\max(1, n)$.
<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$.

beta

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

c

<i>Layout</i> = CblasColMajor	Array, size <i>ldc</i> by <i>n</i> . Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.
<i>Layout</i> = CblasRowMajor	Array, size <i>ldc</i> by <i>m</i> . Before entry, the leading <i>n</i> -by- <i>m</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.

ldc

Specifies the leading dimension of *c* as declared in the calling (sub)program.

<i>Layout</i> = CblasColMajor	<i>ldc</i> must be at least $\max(1, m)$.
<i>Layout</i> = CblasRowMajor	<i>ldc</i> must be at least $\max(1, n)$.

Output Parameters

c

Overwritten by the *m*-by-*n* matrix $(\alpha * \text{op}(A) * \text{op}(B) + \beta * C)$.

Example

For examples of routine usage, see these code examples in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- `cblas_hgemm`: `examples\cblas\source\cblas_hgemmx.c`
- `cblas_sgemm`: `examples\cblas\source\cblas_sgemmx.c`
- `cblas_dgemm`: `examples\cblas\source\cblas_dgemmx.c`
- `cblas_cgemm`: `examples\cblas\source\cblas_cgemmx.c`
- `cblas_zgemm`: `examples\cblas\source\cblas_zgemmx.c`

`cblas_?hemm`

Computes a matrix-matrix product where one input matrix is Hermitian.

Syntax

```
void cblas_chemm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT
lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
```



```
void cblas_zhemm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT
lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The `?hemm` routines compute a scalar-matrix-matrix product using a Hermitian matrix A and a general matrix B and add the result to a scalar-matrix product using a general matrix C . The operation is defined as

$$C := \alpha A * B + \beta C$$

or

$$C := \alpha B * A + \beta C$$

where:

α and β are scalars,

A is a Hermitian matrix,

B and C are m -by- n matrices.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>side</i>	Specifies whether the Hermitian matrix A appears on the left or right in the operation as follows: if $side = \text{CblasLeft}$, then $C := \alpha A * B + \beta C$; if $side = \text{CblasRight}$, then $C := \alpha B * A + \beta C$.
<i>uplo</i>	Specifies whether the upper or lower triangular part of the Hermitian matrix A is used: If $uplo = \text{CblasUpper}$, then the upper triangular part of the Hermitian matrix A is used. If $uplo = \text{CblasLower}$, then the low triangular part of the Hermitian matrix A is used.
<i>m</i>	Specifies the number of rows of the matrix C . The value of m must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix C . The value of n must be at least zero.
<i>alpha</i>	Specifies the scalar α .
<i>a</i>	Array, size $lda * ka$, where ka is m when $side = \text{CblasLeft}$ and is n otherwise. Before entry with $side = \text{CblasLeft}$, the m -by- m part of the array a must contain the Hermitian matrix, such that when $uplo = \text{CblasUpper}$, the leading m -by- m upper triangular part of the array a must

contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of a is not referenced, and when `uplo = CblasLower`, the leading m -by- m lower triangular part of the array a must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of a is not referenced.

Before entry with `side = CblasRight`, the n -by- n part of the array a must contain the Hermitian matrix, such that when `uplo = CblasUpper`, the leading n -by- n upper triangular part of the array a must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of a is not referenced, and when `uplo = CblasLower`, the leading n -by- n lower triangular part of the array a must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of a is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

<code>lda</code>	Specifies the leading dimension of a as declared in the calling (sub) program. When <code>side = CblasLeft</code> then <code>lda</code> must be at least $\max(1, m)$, otherwise <code>lda</code> must be at least $\max(1, n)$.
<code>b</code>	For <code>Layout = CblasColMajor</code> : array, size <code>ldb*n</code> . The leading m -by- n part of the array b must contain the matrix B . For <code>Layout = CblasRowMajor</code> : array, size <code>ldb*m</code> . The leading n -by- m part of the array b must contain the matrix B .
<code>ldb</code>	Specifies the leading dimension of b as declared in the calling (sub)program. When <code>Layout = CblasColMajor</code> , <code>ldb</code> must be at least $\max(1, m)$; otherwise, <code>ldb</code> must be at least $\max(1, n)$.
<code>beta</code>	Specifies the scalar β . When β is supplied as zero, then c need not be set on input.
<code>c</code>	For <code>Layout = CblasColMajor</code> : array, size <code>ldc*n</code> . Before entry, the leading m -by- n part of the array c must contain the matrix C , except when β is zero, in which case c need not be set on entry. For <code>Layout = CblasRowMajor</code> : array, size <code>ldc*m</code> . Before entry, the leading n -by- m part of the array c must contain the matrix C , except when β is zero, in which case c need not be set on entry.
<code>ldc</code>	Specifies the leading dimension of c as declared in the calling (sub)program. When <code>Layout = CblasColMajor</code> , <code>ldc</code> must be at least $\max(1, m)$; otherwise, <code>ldc</code> must be at least $\max(1, n)$.

Output Parameters

<code>c</code>	Overwritten by the m -by- n updated matrix.
----------------	---

`cblas_?herk`

Performs a Hermitian rank- k update.

Syntax

```
void cblas_cherk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const float alpha, const void
*a, const MKL_INT lda, const float beta, void *c, const MKL_INT ldc);
```

```
void cblas_zherk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const double alpha, const void
*a, const MKL_INT lda, const double beta, void *c, const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The `zherk` routines perform a rank- k matrix-matrix operation using a general matrix A and a Hermitian matrix C . The operation is defined as:

$$C := \alpha A A^H + \beta C,$$

or

$$C := \alpha A^H A + \beta C,$$

where:

α and β are real scalars,

C is an n -by- n Hermitian matrix,

A is an n -by- k matrix in the first case and a k -by- n matrix in the second case.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array c is used. If $uplo = \text{CblasUpper}$, then the upper triangular part of the array c is used. If $uplo = \text{CblasLower}$, then the low triangular part of the array c is used.
<i>trans</i>	Specifies the operation: if $trans = \text{CblasNoTrans}$, then $C := \alpha A A^H + \beta C$; if $trans = \text{CblasConjTrans}$, then $C := \alpha A^H A + \beta C$.
<i>n</i>	Specifies the order of the matrix C . The value of n must be at least zero.
<i>k</i>	With $trans = \text{CblasNoTrans}$, k specifies the number of columns of the matrix A , and with $trans = \text{CblasConjTrans}$, k specifies the number of rows of the matrix A . The value of k must be at least zero.
<i>alpha</i>	Specifies the scalar α .

a

	$trans = \text{CblasNoTrans}$	$trans = \text{CblasConjTrans}$
$Layout = \text{CblasColMajor}$	Array, size $lda * k$.	Array, size $lda * n$. Before entry, the leading k -by- n part of the array a must contain the matrix A .

	Before entry, the leading n -by- k part of the array a must contain the matrix A .	
$Layout =$ $CblasRowMajor$	Array, size $lda*n$. Before entry, the leading k -by- n part of the array a must contain the matrix A .	Array, size $lda*k$. Before entry, the leading n -by- k part of the array a must contain the matrix A .

 lda

	$trans=CblasNoTrans$	$trans=CblasConjTrans$
$Layout =$ $CblasColMajor$	lda must be at least $\max(1, n)$.	lda must be at least $\max(1, k)$
$Layout =$ $CblasRowMajor$	lda must be at least $\max(1, k)$	lda must be at least $\max(1, n)$.

 $beta$ Specifies the scalar $beta$. c Array, size ldc by n .

Before entry with $uplo = CblasUpper$, the leading n -by- n upper triangular part of the array c must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of c is not referenced.

Before entry with $uplo = CblasLower$, the leading n -by- n lower triangular part of the array c must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of c is not referenced.

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

 ldc

Specifies the leading dimension of c as declared in the calling (sub)program. The value of ldc must be at least $\max(1, n)$.

Output Parameters

 c

With $uplo = CblasUpper$, the upper triangular part of the array c is overwritten by the upper triangular part of the updated matrix.

With $uplo = CblasLower$, the lower triangular part of the array c is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

cblas_?her2k

Performs a Hermitian rank-2k update.

Syntax

```
void cblas_cher2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *b, const MKL_INT ldb, const float beta, void *c,
const MKL_INT ldc);
```

```
void cblas_zher2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *b, const MKL_INT ldb, const double beta, void *c,
const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The `cher2k` routines perform a rank-2k matrix-matrix operation using general matrices A and B and a Hermitian matrix C . The operation is defined as

$$C := \alpha A B^H + \text{conjg}(\alpha) B A^H + \beta C$$

or

$$C := \alpha A^H B + \text{conjg}(\alpha) B^H A + \beta C$$

where:

α is a scalar and β is a real scalar.

C is an n -by- n Hermitian matrix.

A and B are n -by- k matrices in the first case and k -by- n matrices in the second case.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array c is used. If $uplo = \text{CblasUpper}$, then the upper triangular of the array c is used. If $uplo = \text{CblasLower}$, then the low triangular of the array c is used.
<i>trans</i>	Specifies the operation: if $trans = \text{CblasNoTrans}$, then $C := \alpha A B^H + \alpha B A^H + \beta C$; if $trans = \text{CblasConjTrans}$, then $C := \alpha A^H B + \alpha B^H A + \beta C$.
<i>n</i>	Specifies the order of the matrix C . The value of n must be at least zero.
<i>k</i>	With $trans = \text{CblasNoTrans}$ specifies the number of columns of the matrix A , and with $trans = \text{CblasConjTrans}$, k specifies the number of rows of the matrix A . The value of k must be at least equal to zero.
<i>alpha</i>	Specifies the scalar α .

a

	<i>trans</i> =CblasNoTrans	<i>trans</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size <i>lda</i> * <i>n</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>Layout</i> = CblasRowMajor	Array, size <i>lda</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .

lda

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

	<i>trans</i> =CblasNoTrans	<i>trans</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1, n)$.	<i>lda</i> must be at least $\max(1, k)$
<i>Layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1, k)$	<i>lda</i> must be at least $\max(1, n)$.

beta

Specifies the scalar *beta*.

b

	<i>trans</i> =CblasNoTrans	<i>trans</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	Array, size <i>ldb</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>ldb</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>Layout</i> = CblasRowMajor	Array, size <i>lda</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .

ldb

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

	<i>trans</i> =CblasNoTrans	<i>trans</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$

<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, k)$	<i>ldb</i> must be at least $\max(1, n)$.
----------------------------------	---	---

*c*Array, size *ldc* by *n*.

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

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

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

ldc

Specifies the leading dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least $\max(1, n)$.

Output Parameters

c

With *uplo* = CblasUpper, the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = CblasLower, the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

cblas_?symm

Computes a matrix-matrix product where one input matrix is symmetric.

Syntax

```
void cblas_ssymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const MKL_INT m, const MKL_INT n, const float alpha, const float *a, const
MKL_INT lda, const float *b, const MKL_INT ldb, const float beta, float *c, const
MKL_INT ldc);
```

```
void cblas_dsymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const MKL_INT m, const MKL_INT n, const double alpha, const double *a, const
MKL_INT lda, const double *b, const MKL_INT ldb, const double beta, double *c, const
MKL_INT ldc);
```

```
void cblas_csymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT
lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
```

```
void cblas_zsymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT
lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The ?`symm` routines compute a scalar-matrix-matrix product with one symmetric matrix and add the result to a scalar-matrix product. The operation is defined as

```
C := alpha*A*B + beta*C,
```

or

```
C := alpha*B*A + beta*C,
```

where:

alpha and *beta* are scalars,

A is a symmetric matrix,

B and *C* are *m*-by-*n* matrices.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>side</i>	Specifies whether the symmetric matrix <i>A</i> appears on the left or right in the operation: if <i>side</i> = CblasLeft, then <i>C</i> := <i>alpha</i> * <i>A</i> * <i>B</i> + <i>beta</i> * <i>C</i> ; if <i>side</i> = CblasRight, then <i>C</i> := <i>alpha</i> * <i>B</i> * <i>A</i> + <i>beta</i> * <i>C</i> .
<i>uplo</i>	Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is used: if <i>uplo</i> = CblasUpper, then the upper triangular part is used; if <i>uplo</i> = CblasLower, then the lower triangular part is used.
<i>m</i>	Specifies the number of rows of the matrix <i>C</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix <i>C</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>a</i>	Array, size <i>lda</i> * <i>ka</i> , where <i>ka</i> is <i>m</i> when <i>side</i> = CblasLeft and is <i>n</i> otherwise. Before entry with <i>side</i> = CblasLeft, the <i>m</i> -by- <i>m</i> part of the array <i>a</i> must contain the symmetric matrix, such that when <i>uplo</i> = CblasUpper, the leading <i>m</i> -by- <i>m</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>a</i> is not referenced, and when <i>side</i> = CblasLeft, the leading <i>m</i> -by- <i>m</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>a</i> is not referenced.

Before entry with *side* = CblasRight, the *n*-by-*n* part of the array *a* must contain the symmetric matrix, such that when *uplo* = CblasUpper array *a* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *a* is not referenced, and when *side* = CblasLeft, the leading *n*-by-*n* lower triangular part of the array *a* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *a* is not referenced.

<i>lda</i>	Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>side</i> = CblasLeft then <i>lda</i> must be at least $\max(1, m)$, otherwise <i>lda</i> must be at least $\max(1, n)$.
<i>b</i>	For <i>Layout</i> = CblasColMajor: array, size <i>ldb</i> * <i>n</i> . The leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> . For <i>Layout</i> = CblasRowMajor: array, size <i>ldb</i> * <i>m</i> . The leading <i>n</i> -by- <i>m</i> part of the array <i>b</i> must contain the matrix <i>B</i>
<i>ldb</i>	Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. When <i>Layout</i> = CblasColMajor, <i>ldb</i> must be at least $\max(1, m)$; otherwise, <i>ldb</i> must be at least $\max(1, n)$.
<i>beta</i>	Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <i>c</i> need not be set on input.
<i>c</i>	For <i>Layout</i> = CblasColMajor: array, size <i>ldc</i> * <i>n</i> . Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is zero, in which case <i>c</i> need not be set on entry. For <i>Layout</i> = CblasRowMajor: array, size <i>ldc</i> * <i>m</i> . Before entry, the leading <i>n</i> -by- <i>m</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is zero, in which case <i>c</i> need not be set on entry.
<i>ldc</i>	Specifies the leading dimension of <i>c</i> as declared in the calling (sub)program. When <i>Layout</i> = CblasColMajor, <i>ldc</i> must be at least $\max(1, m)$; otherwise, <i>ldc</i> must be at least $\max(1, n)$.

Output Parameters

<i>c</i>	Overwritten by the <i>m</i> -by- <i>n</i> updated matrix.
----------	---

cblas_?syrk

Performs a symmetric rank-k update.

Syntax

```
void cblas_ssyrrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const float alpha, const float
*a, const MKL_INT lda, const float beta, float *c, const MKL_INT ldc);

void cblas_dsyrrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const double alpha, const
double *a, const MKL_INT lda, const double beta, double *c, const MKL_INT ldc);

void cblas_csyrrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *beta, void *c, const MKL_INT ldc);
```

```
void cblas_zsyrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *beta, void *c, const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The ?syrk routines perform a rank- k matrix-matrix operation for a symmetric matrix C using a general matrix A . The operation is defined as:

$$C := \alpha A A' + \beta C,$$

or

$$C := \alpha A' A + \beta C,$$

where:

α and β are scalars,

C is an n -by- n symmetric matrix,

A is an n -by- k matrix in the first case and a k -by- n matrix in the second case.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array c is used. If $uplo = \text{CblasUpper}$, then the upper triangular part of the array c is used. If $uplo = \text{CblasLower}$, then the low triangular part of the array c is used.
<i>trans</i>	Specifies the operation: if $trans = \text{CblasNoTrans}$, then $C := \alpha A A' + \beta C$; if $trans = \text{CblasTrans}$, then $C := \alpha A' A + \beta C$; if $trans = \text{CblasConjTrans}$, then $C := \alpha A' A + \beta C$.
<i>n</i>	Specifies the order of the matrix C . The value of n must be at least zero.
<i>k</i>	On entry with $trans = \text{CblasNoTrans}$, k specifies the number of columns of the matrix a , and on entry with $trans = \text{CblasTrans}$ or $trans = \text{CblasConjTrans}$, k specifies the number of rows of the matrix a . The value of k must be at least zero.
<i>alpha</i>	Specifies the scalar α .
<i>a</i>	Array, size $lda * ka$, where ka is k when $trans = \text{CblasNoTrans}$, and is n otherwise. Before entry with $trans = \text{CblasNoTrans}$, the leading n -by- k part of the array a must contain the matrix A , otherwise the leading k -by- n part of the array a must contain the matrix A .

	<i>trans</i> =CblasNoTrans	<i>trans</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size <i>lda</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>Layout</i> = CblasRowMajor	Array, size <i>lda</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .

lda

	<i>trans</i> =CblasNoTrans	<i>trans</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1, n)$.	<i>lda</i> must be at least $\max(1, k)$
<i>Layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1, k)$	<i>lda</i> must be at least $\max(1, n)$.

*beta*Specifies the scalar *beta*.*c*

Array, size *ldc** *n*. Before entry with *uplo* = CblasUpper, the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *c* is not referenced.

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

ldc

Specifies the leading dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least $\max(1, n)$.

Output Parameters

c

With *uplo* = CblasUpper, the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = CblasLower, the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

cblas_?syr2k

Performs a symmetric rank-2k update.

Syntax

```
void cblas_ssyr2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const float alpha, const float
*a, const MKL_INT lda, const float *b, const MKL_INT ldb, const float beta, float *c,
const MKL_INT ldc);
```

```

void cblas_dsyr2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const double alpha, const
double *a, const MKL_INT lda, const double *b, const MKL_INT ldb, const double beta,
double *c, const MKL_INT ldc);

void cblas_csyr2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c,
const MKL_INT ldc);

void cblas_zsyr2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c,
const MKL_INT ldc);

```

Include Files

- mkl.h

Description

The ?syr2k routines perform a rank-2k matrix-matrix operation for a symmetric matrix C using general matrices A and B . The operation is defined as:

$$C := \alpha A B' + \alpha B A' + \beta C,$$

or

$$C := \alpha A' B + \alpha B' A + \beta C,$$

where:

α and β are scalars,

C is an n -by- n symmetric matrix,

A and B are n -by- k matrices in the first case, and k -by- n matrices in the second case.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array c is used. If $uplo = \text{CblasUpper}$, then the upper triangular part of the array c is used. If $uplo = \text{CblasLower}$, then the low triangular part of the array c is used.
<i>trans</i>	Specifies the operation: if $trans = \text{CblasNoTrans}$, then $C := \alpha A B' + \alpha B A' + \beta C$; if $trans = \text{CblasTrans}$, then $C := \alpha A' B + \alpha B' A + \beta C$; if $trans = \text{CblasConjTrans}$, then $C := \alpha A' B + \alpha B' A + \beta C$.
<i>n</i>	Specifies the order of the matrix C . The value of n must be at least zero.

k On entry with *trans*=CblasNoTrans, *k* specifies the number of columns of the matrices *A* and *B*, and on entry with *trans*=CblasTrans or *trans*=CblasConjTrans, *k* specifies the number of rows of the matrices *A* and *B*. The value of *k* must be at least zero.

alpha Specifies the scalar *alpha*.

a

	<i>trans</i> =CblasNoTrans	<i>trans</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size <i>lda</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>Layout</i> = CblasRowMajor	Array, size <i>lda</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .

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

	<i>trans</i> =CblasNoTrans	<i>trans</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1, n)$.	<i>lda</i> must be at least $\max(1, k)$
<i>Layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1, k)$	<i>lda</i> must be at least $\max(1, n)$.

b

	<i>trans</i> =CblasNoTrans	<i>trans</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	Array, size <i>ldb</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>ldb</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>Layout</i> = CblasRowMajor	Array, size <i>lda</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .

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

	<i>trans</i> =CblasNoTrans	<i>trans</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$
<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, k)$	<i>ldb</i> must be at least $\max(1, n)$.

*beta*Specifies the scalar *beta*.*c*

Array, size *ldc** *n*. Before entry with *uplo* = CblasUpper, the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *c* is not referenced.

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

ldc

Specifies the leading dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least $\max(1, n)$.

Output Parameters

c

With *uplo* = CblasUpper, the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = CblasLower, the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

cblas_?trmm

Computes a matrix-matrix product where one input matrix is triangular.

Syntax

```
void cblas_strmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const float alpha, const float *a, const MKL_INT lda, float *b, const
MKL_INT ldb);
```

```
void cblas_dtrmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const double alpha, const double *a, const MKL_INT lda, double *b, const
MKL_INT ldb);
```

```
void cblas_ctrmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT
ldb);
```

```
void cblas_ztrmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT
ldb);
```

Include Files

- `mk1.h`

Description

The `?trmm` routines compute a scalar-matrix-matrix product with one triangular matrix. The operation is defined as

$$B := \alpha * \text{op}(A) * B$$

or

$$B := \alpha * B * \text{op}(A)$$

where:

α is a scalar,

B is an m -by- n matrix,

A is a unit, or non-unit, upper or lower triangular matrix

$\text{op}(A)$ is one of $\text{op}(A) = A$, or $\text{op}(A) = A'$, or $\text{op}(A) = \text{conjg}(A')$.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>side</i>	Specifies whether $\text{op}(A)$ appears on the left or right of B in the operation: if $\text{side} = \text{CblasLeft}$, then $B := \alpha * \text{op}(A) * B$; if $\text{side} = \text{CblasRight}$, then $B := \alpha * B * \text{op}(A)$.
<i>uplo</i>	Specifies whether the matrix A is upper or lower triangular. $\text{uplo} = \text{CblasUpper}$ if $\text{uplo} = \text{CblasLower}$, then the matrix is low triangular.
<i>transa</i>	Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if $\text{transa} = \text{CblasNoTrans}$, then $\text{op}(A) = A$; if $\text{transa} = \text{CblasTrans}$, then $\text{op}(A) = A'$; if $\text{transa} = \text{CblasConjTrans}$, then $\text{op}(A) = \text{conjg}(A')$.
<i>diag</i>	Specifies whether the matrix A is unit triangular: if $\text{diag} = \text{CblasUnit}$ then the matrix is unit triangular; if $\text{diag} = \text{CblasNonUnit}$, then the matrix is not unit triangular.
<i>m</i>	Specifies the number of rows of B . The value of m must be at least zero.
<i>n</i>	Specifies the number of columns of B . The value of n must be at least zero.
<i>alpha</i>	Specifies the scalar α . When α is zero, then a is not referenced and b need not be set before entry.

<i>a</i>	<p>Array, size <i>lda</i> by <i>k</i>, where <i>k</i> is <i>m</i> when <i>side</i> = CblasLeft and is <i>n</i> when <i>side</i> = CblasRight. Before entry with <i>uplo</i> = CblasUpper, the leading <i>k</i> by <i>k</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced.</p> <p>Before entry with <i>uplo</i> = CblasLower, the leading <i>k</i> by <i>k</i> lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced.</p> <p>When <i>diag</i> = CblasUnit, the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.</p>
<i>lda</i>	<p>Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>side</i> = CblasLeft, then <i>lda</i> must be at least $\max(1, m)$, when <i>side</i> = CblasRight, then <i>lda</i> must be at least $\max(1, n)$.</p>
<i>b</i>	<p>For <i>Layout</i> = CblasColMajor: array, size <i>ldb</i>*<i>n</i>. Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.</p> <p>For <i>Layout</i> = CblasRowMajor: array, size <i>ldb</i>*<i>m</i>. Before entry, the leading <i>n</i>-by-<i>m</i> part of the array <i>b</i> must contain the matrix <i>B</i>.</p>
<i>ldb</i>	<p>Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. When <i>Layout</i> = CblasColMajor, <i>ldb</i> must be at least $\max(1, m)$; otherwise, <i>ldb</i> must be at least $\max(1, n)$.</p>

Output Parameters

<i>b</i>	Overwritten by the transformed matrix.
----------	--

cblas_?trsm

Solves a triangular matrix equation.

Syntax

```
void cblas_strsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const float alpha, const float *a, const MKL_INT lda, float *b, const
MKL_INT ldb);
```

```
void cblas_dtrsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const double alpha, const double *a, const MKL_INT lda, double *b, const
MKL_INT ldb);
```

```
void cblas_ctrsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT
ldb);
```

```
void cblas_ztrsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT
ldb);
```

Include Files

- mkl.h

Description

The `?trsm` routines solve one of the following matrix equations:

$$\text{op}(A) * X = \alpha * B,$$

or

$$X * \text{op}(A) = \alpha * B,$$

where:

α is a scalar,

X and B are m -by- n matrices,

A is a unit, or non-unit, upper or lower triangular matrix, and

$\text{op}(A)$ is one of $\text{op}(A) = A$, or $\text{op}(A) = A'$, or $\text{op}(A) = \text{conjg}(A')$.

The matrix B is overwritten by the solution matrix X .

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>side</i>	Specifies whether $\text{op}(A)$ appears on the left or right of X in the equation: if $side = \text{CblasLeft}$, then $\text{op}(A) * X = \alpha * B$; if $side = \text{CblasRight}$, then $X * \text{op}(A) = \alpha * B$.
<i>uplo</i>	Specifies whether the matrix A is upper or lower triangular. $uplo = \text{CblasUpper}$ if $uplo = \text{CblasLower}$, then the matrix is low triangular.
<i>transa</i>	Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if $transa = \text{CblasNoTrans}$, then $\text{op}(A) = A$; if $transa = \text{CblasTrans}$; if $transa = \text{CblasConjTrans}$, then $\text{op}(A) = \text{conjg}(A')$.
<i>diag</i>	Specifies whether the matrix A is unit triangular: if $diag = \text{CblasUnit}$ then the matrix is unit triangular; if $diag = \text{CblasNonUnit}$, then the matrix is not unit triangular.
<i>m</i>	Specifies the number of rows of B . The value of m must be at least zero.
<i>n</i>	Specifies the number of columns of B . The value of n must be at least zero.
<i>alpha</i>	Specifies the scalar α . When α is zero, then a is not referenced and b need not be set before entry.
<i>a</i>	Array, size $lda * k$, where k is m when $side = \text{CblasLeft}$ and is n when $side = \text{CblasRight}$. Before entry with $uplo = \text{CblasUpper}$, the leading k by k upper triangular part of the array a must contain the upper triangular matrix and the strictly lower triangular part of a is not referenced.

Before entry with `uplo = CblasLower` lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced.

When `diag = CblasUnit`, the diagonal elements of *a* are not referenced either, but are assumed to be unity.

<i>lda</i>	Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <code>side = CblasLeft</code> , then <i>lda</i> must be at least $\max(1, m)$, when <code>side = CblasRight</code> , then <i>lda</i> must be at least $\max(1, n)$.
<i>b</i>	For <code>Layout = CblasColMajor</code> : array, size <i>ldb</i> * <i>n</i> . Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> . For <code>Layout = CblasRowMajor</code> : array, size <i>ldb</i> * <i>m</i> . Before entry, the leading <i>n</i> -by- <i>m</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>ldb</i>	Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. When <code>Layout = CblasColMajor</code> , <i>ldb</i> must be at least $\max(1, m)$; otherwise, <i>ldb</i> must be at least $\max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

Sparse BLAS Level 1 Routines

This section describes Sparse BLAS Level 1, an extension of BLAS Level 1 included in the Intel® oneAPI Math Kernel Library beginning with the Intel® oneAPI Math Kernel Library (oneMKL) release 2.1. Sparse BLAS Level 1 is a group of routines and functions that perform a number of common vector operations on sparse vectors stored in compressed form.

Sparse vectors are those in which the majority of elements are zeros. Sparse BLAS routines and functions are specially implemented to take advantage of vector sparsity. This allows you to achieve large savings in computer time and memory. If *nz* is the number of non-zero vector elements, the computer time taken by Sparse BLAS operations will be $O(nz)$.

Vector Arguments

Compressed sparse vectors. Let *a* be a vector stored in an array, and assume that the only non-zero elements of *a* are the following:

```
a[k1], a[k2], a[k3] . . . a[knz],
```

where *nz* is the total number of non-zero elements in *a*.

In Sparse BLAS, this vector can be represented in compressed form by two arrays, *x* (values) and *indx* (indices). Each array has *nz* elements:

```
x[0]=a[k1], x[1]=a[k2], . . . x[nz-1]= a[knz],
```

```
indx[0]=k1, indx[1]=k2, . . . indx[nz-1]= knz.
```

Thus, a sparse vector is fully determined by the triple (*nz*, *x*, *indx*). If you pass a negative or zero value of *nz* to Sparse BLAS, the subroutines do not modify any arrays or variables.

Full-storage vectors. Sparse BLAS routines can also use a vector argument fully stored in a single array (a full-storage vector). If *y* is a full-storage vector, its elements must be stored contiguously: the first element in *y*[0], the second in *y*[1], and so on. This corresponds to an increment *incy* = 1 in BLAS Level 1. No increment value for full-storage vectors is passed as an argument to Sparse BLAS routines or functions.

Naming Conventions for Sparse BLAS Routines

Similar to BLAS, the names of Sparse BLAS subprograms have prefixes that determine the data type involved: *s* and *d* for single- and double-precision real; *c* and *z* for single- and double-precision complex respectively.

If a Sparse BLAS routine is an extension of a "dense" one, the subprogram name is formed by appending the suffix *i* (standing for *indexed*) to the name of the corresponding "dense" subprogram. For example, the Sparse BLAS routine `saxpyi` corresponds to the BLAS routine `saxpy`, and the Sparse BLAS function `cdotci` corresponds to the BLAS function `cdotc`.

Routines and Data Types

Routines and data types supported in the Intel® oneAPI Math Kernel Library (oneMKL) implementation of Sparse BLAS are listed in Table "Sparse BLAS Routines and Their Data Types".

Sparse BLAS Routines and Their Data Types

Routine/ Function	Data Types	Description
<code>cblas_?axpyi</code>	<i>s, d, c, z</i>	Scalar-vector product plus vector (routines)
<code>cblas_?doti</code>	<i>s, d</i>	Dot product (functions)
<code>cblas_?dotci</code>	<i>c, z</i>	Complex dot product conjugated (functions)
<code>cblas_?dotui</code>	<i>c, z</i>	Complex dot product unconjugated (functions)
<code>cblas_?gthr</code>	<i>s, d, c, z</i>	Gathering a full-storage sparse vector into compressed form <i>nz, x, indx</i> (routines)
<code>cblas_?gthrz</code>	<i>s, d, c, z</i>	Gathering a full-storage sparse vector into compressed form and assigning zeros to gathered elements in the full-storage vector (routines)
<code>cblas_?roti</code>	<i>s, d</i>	Givens rotation (routines)
<code>cblas_?sctr</code>	<i>s, d, c, z</i>	Scattering a vector from compressed form to full-storage form (routines)

BLAS Level 1 Routines That Can Work With Sparse Vectors

The following BLAS Level 1 routines will give correct results when you pass to them a compressed-form array *x* (with the increment `incx=1`):

<code>cblas_?asum</code>	sum of absolute values of vector elements
<code>cblas_?copy</code>	copying a vector
<code>cblas_?nrm2</code>	Euclidean norm of a vector
<code>cblas_?scal</code>	scaling a vector
<code>cblas_i?amax</code>	index of the element with the largest absolute value for real flavors, or the largest sum $ \text{Re}(x[i]) + \text{Im}(x[i]) $ for complex flavors.
<code>cblas_i?amin</code>	index of the element with the smallest absolute value for real flavors, or the smallest sum $ \text{Re}(x[i]) + \text{Im}(x[i]) $ for complex flavors.

The result *i* returned by `i?amax` and `i?amin` should be interpreted as index in the compressed-form array, so that the largest (smallest) value is `x[i-1]`; the corresponding index in full-storage array is `indx[i-1]`.

You can also call `cblas_?rotg` to compute the parameters of Givens rotation and then pass these parameters to the Sparse BLAS routines `cblas_?roti`.

cblas_?axpyi

Adds a scalar multiple of compressed sparse vector to a full-storage vector.

Syntax

```
void cblas_saxpyi (const MKL_INT nz, const float a, const float *x, const MKL_INT
*indx, float *y);

void cblas_daxpyi (const MKL_INT nz, const double a, const double *x, const MKL_INT
*indx, double *y);

void cblas_caxpyi (const MKL_INT nz, const void *a, const void *x, const MKL_INT *indx,
void *y);

void cblas_zaxpyi (const MKL_INT nz, const void *a, const void *x, const MKL_INT *indx,
void *y);
```

Include Files

- `mkl.h`

Description

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

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

where:

a is a scalar,

x is a sparse vector stored in compressed form,

y is a vector in full storage form.

The `?axpyi` routines reference or modify only the elements of *y* whose indices are listed in the array *indx*.

The values in *indx* must be distinct.

Input Parameters

<i>nz</i>	The number of elements in <i>x</i> and <i>indx</i> .
<i>a</i>	Specifies the scalar <i>a</i> .
<i>x</i>	Array, size at least <i>nz</i> .
<i>indx</i>	Specifies the indices for the elements of <i>x</i> . Array, size at least <i>nz</i> .
<i>y</i>	Array, size at least <code>max(indx[i])</code> .

Output Parameters

<i>y</i>	Contains the updated vector <i>y</i> .
----------	--

cblas_?doti

Computes the dot product of a compressed sparse real vector by a full-storage real vector.

Syntax

```
float cblas_sdoti (const MKL_INT nz, const float *x, const MKL_INT *indx, const float *y);
```

```
double cblas_ddoti (const MKL_INT nz, const double *x, const MKL_INT *indx, const double *y);
```

Include Files

- mkl.h

Description

The ?doti routines return the dot product of x and y defined as

```
res = x[0]*y[indx[0]] + x[1]*y[indx[1]] + ... + x[nz-1]*y[indx[nz-1]]
```

where the triple $(nz, x, indx)$ defines a sparse real vector stored in compressed form, and y is a real vector in full storage form. The functions reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

nz	The number of elements in x and $indx$.
x	Array, size at least nz .
$indx$	Specifies the indices for the elements of x . Array, size at least nz .
y	Array, size at least $\max(indx[i])$.

Output Parameters

res	Contains the dot product of x and y , if nz is positive. Otherwise, res contains 0.
-------	---

cblas_?dotci

Computes the conjugated dot product of a compressed sparse complex vector with a full-storage complex vector.

Syntax

```
void cblas_cdotci_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void *y, void *dotui);
```

```
void cblas_zdotci_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void *y, void *dotui);
```

Include Files

- mkl.h

Description

The `?dotci` routines return the dot product of x and y defined as

```
conjg(x[0])*y[indx[0]] + ... + conjg(x[nz-1])*y[indx[nz-1]]
```

where the triple $(nz, x, indx)$ defines a sparse complex vector stored in compressed form, and y is a real vector in full storage form. The functions reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

nz	The number of elements in x and $indx$.
x	Array, size at least nz .
$indx$	Specifies the indices for the elements of x . Array, size at least nz .
y	Array, size at least $\max(indx[i])$.

Output Parameters

$dotui$	Contains the conjugated dot product of x and y , if nz is positive. Otherwise, it contains 0.
---------	---

cblas_?dotui

Computes the dot product of a compressed sparse complex vector by a full-storage complex vector.

Syntax

```
void cblas_cdotui_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void *y, void *dotui);
```

```
void cblas_zdotui_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void *y, void *dotui);
```

Include Files

- mkl.h

Description

The `?dotui` routines return the dot product of x and y defined as

```
res = x[0]*y[indx[0]] + x[1]*y[indx[1]] +...+ x[nz - 1]*y[indx[nz - 1]]
```

where the triple $(nz, x, indx)$ defines a sparse complex vector stored in compressed form, and y is a real vector in full storage form. The functions reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

<i>nz</i>	The number of elements in <i>x</i> and <i>indx</i> .
<i>x</i>	Array, size at least <i>nz</i> .
<i>indx</i>	Specifies the indices for the elements of <i>x</i> . Array, size at least <i>nz</i> .
<i>y</i>	Array, size at least $\max(\text{indx}[i])$.

Output Parameters

<i>dotui</i>	Contains the dot product of <i>x</i> and <i>y</i> , if <i>nz</i> is positive. Otherwise, <i>res</i> contains 0.
--------------	---

cblas_?gthr

Gathers a full-storage sparse vector's elements into compressed form.

Syntax

```
void cblas_sgthr (const MKL_INT nz, const float *y, float *x, const MKL_INT *indx);
void cblas_dgthr (const MKL_INT nz, const double *y, double *x, const MKL_INT *indx);
void cblas_cgthr (const MKL_INT nz, const void *y, void *x, const MKL_INT *indx);
void cblas_zgthr (const MKL_INT nz, const void *y, void *x, const MKL_INT *indx);
```

Include Files

- `mkl.h`

Description

The `?gthr` routines gather the specified elements of a full-storage sparse vector *y* into compressed form(*nz*, *x*, *indx*). The routines reference only the elements of *y* whose indices are listed in the array *indx*:

$x[i] = y[indx[i]], \text{ for } i=0,1,\dots, nz-1.$

Input Parameters

<i>nz</i>	The number of elements of <i>y</i> to be gathered.
<i>indx</i>	Specifies indices of elements to be gathered. Array, size at least <i>nz</i> .
<i>y</i>	Array, size at least $\max(\text{indx}[i])$.

Output Parameters

<i>x</i>	Array, size at least <i>nz</i> . Contains the vector converted to the compressed form.
----------	---

cblas_?gthrz

Gathers a sparse vector's elements into compressed form, replacing them by zeros.

Syntax

```
void cblas_sgthrz (const MKL_INT nz, float *y, float *x, const MKL_INT *indx);
void cblas_dgthrz (const MKL_INT nz, double *y, double *x, const MKL_INT *indx);
void cblas_cgthrz (const MKL_INT nz, void *y, void *x, const MKL_INT *indx);
void cblas_zgthrz (const MKL_INT nz, void *y, void *x, const MKL_INT *indx);
```

Include Files

- mkl.h

Description

The `?gthrz` routines gather the elements with indices specified by the array `indx` from a full-storage vector `y` into compressed form (`nz`, `x`, `indx`) and overwrite the gathered elements of `y` by zeros. Other elements of `y` are not referenced or modified (see also `?gthr`).

Input Parameters

<code>nz</code>	The number of elements of <code>y</code> to be gathered.
<code>indx</code>	Specifies indices of elements to be gathered. Array, size at least <code>nz</code> .
<code>y</code>	Array, size at least <code>max(indx[i])</code> .

Output Parameters

<code>x</code>	Array, size at least <code>nz</code> . Contains the vector converted to the compressed form.
<code>y</code>	The updated vector <code>y</code> .

cblas_?roti

Applies Givens rotation to sparse vectors one of which is in compressed form.

Syntax

```
void cblas_sroti (const MKL_INT nz, float *x, const MKL_INT *indx, float *y, const float c, const float s);
void cblas_droti (const MKL_INT nz, double *x, const MKL_INT *indx, double *y, const double c, const double s);
```

Include Files

- mkl.h

Description

The `?roti` routines apply the Givens rotation to elements of two real vectors, x (in compressed form nz , x , $indx$) and y (in full storage form):

$$x[i] = c*x[i] + s*y[indx[i]]$$

$$y[indx[i]] = c*y[indx[i]] - s*x[i]$$

The routines reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

nz	The number of elements in x and $indx$.
x	Array, size at least nz .
$indx$	Specifies the indices for the elements of x . Array, size at least nz .
y	Array, size at least $\max(indx[i])$.
c	A scalar.
s	A scalar.

Output Parameters

x and y	The updated arrays.
-------------	---------------------

`cblas_?sctr`

Converts compressed sparse vectors into full storage form.

Syntax

```
void cblas_ssctr (const MKL_INT nz, const float *x, const MKL_INT *indx, float *y);
void cblas_dsctr (const MKL_INT nz, const double *x, const MKL_INT *indx, double *y);
void cblas_csctr (const MKL_INT nz, const void *x, const MKL_INT *indx, void *y);
void cblas_zsctr (const MKL_INT nz, const void *x, const MKL_INT *indx, void *y);
```

Include Files

- `mkl.h`

Description

The `?sctr` routines scatter the elements of the compressed sparse vector (nz , x , $indx$) to a full-storage vector y . The routines modify only the elements of y whose indices are listed in the array $indx$:

$$y[indx[i]] = x[i], \text{ for } i=0, 1, \dots, nz-1.$$

Input Parameters

<i>nz</i>	The number of elements of <i>x</i> to be scattered.
<i>indx</i>	Specifies indices of elements to be scattered. Array, size at least <i>nz</i> .
<i>x</i>	Array, size at least <i>nz</i> . Contains the vector to be converted to full-storage form.

Output Parameters

<i>y</i>	Array, size at least $\max(\text{indx}[i])$. Contains the vector <i>y</i> with updated elements.
----------	--

Sparse BLAS Level 2 and Level 3 Routines

NOTE The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines are deprecated. Use the corresponding routine from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface as indicated in the description for each routine.

This section describes Sparse BLAS Level 2 and Level 3 routines included in the Intel® oneAPI Math Kernel Library (oneMKL). Sparse BLAS Level 2 is a group of routines and functions that perform operations between a sparse matrix and dense vectors. Sparse BLAS Level 3 is a group of routines and functions that perform operations between a sparse matrix and dense matrices.

The terms and concepts required to understand the use of the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines are discussed in the [Linear Solvers Basics](#) appendix.

The Sparse BLAS routines can be useful to implement iterative methods for solving large sparse systems of equations or eigenvalue problems. For example, these routines can be considered as building blocks for [Iterative Sparse Solvers based on Reverse Communication Interface \(RCI ISS\)](#).

Intel® oneAPI Math Kernel Library (oneMKL) provides Sparse BLAS Level 2 and Level 3 routines with typical (or conventional) interface similar to the interface used in the NIST* Sparse BLAS library [[Rem05](#)].

Some software packages and libraries (the [PARDISO* Solver](#) used in Intel® oneAPI Math Kernel Library (oneMKL), *Sparskit 2* [[Saad94](#)], the Compaq* Extended Math Library (CXML)[[CXML01](#)]) use different (early) variation of the compressed sparse row (CSR) format and support only Level 2 operations with simplified interfaces. Intel® oneAPI Math Kernel Library (oneMKL) provides an additional set of Sparse BLAS Level 2 routines with similar simplified interfaces. Each of these routines operates only on a matrix of the fixed type.

The routines described in this section support both one-based indexing and zero-based indexing of the input data (see details in the section [One-based and Zero-based Indexing](#)).

Naming Conventions in Sparse BLAS Level 2 and Level 3

Each Sparse BLAS Level 2 and Level 3 routine has a six- or eight-character base name preceded by the prefix `mkl_` or `mkl_cspblas_`.

The routines with typical (conventional) interface have six-character base names in accordance with the template:

```
mkl_<character> <data> <operation>( )
```

The routines with simplified interfaces have eight-character base names in accordance with the templates:

```
mkl_<character> <data> <mtype> <operation>( )
```

for routines with one-based indexing; and

```
mkl_cspblas_<character> <data><mtype><operation>( )
```

for routines with zero-based indexing.

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

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

The *<data>* field indicates the sparse matrix storage format (see section [Sparse Matrix Storage Formats](#)):

coo	coordinate format
csr	compressed sparse row format and its variations
csc	compressed sparse column format and its variations
dia	diagonal format
sky	skyline storage format
bsr	block sparse row format and its variations

The *<operation>* field indicates the type of operation:

mv	matrix-vector product (Level 2)
mm	matrix-matrix product (Level 3)
sv	solving a single triangular system (Level 2)
sm	solving triangular systems with multiple right-hand sides (Level 3)

The field *<mtype>* indicates the matrix type:

ge	sparse representation of a general matrix
sy	sparse representation of the upper or lower triangle of a symmetric matrix
tr	sparse representation of a triangular matrix

Sparse Matrix Storage Formats for Sparse BLAS Routines

The current version of Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines support the following point entry [\[Duff86\]](#) storage formats for sparse matrices:

- *compressed sparse row* format (CSR) and its variations;
- *compressed sparse column* format (CSC);
- *coordinate* format;
- *diagonal* format;
- *skyline* storage format;

and one block entry storage format:

- *block sparse row* format (BSR) and its variations.

For more information see "[Sparse Matrix Storage Formats](#)" in the Appendix "Linear Solvers Basics".

Intel® oneAPI Math Kernel Library (oneMKL) provides auxiliary routines -[matrix converters](#) - that convert sparse matrix from one storage format to another.

Routines and Supported Operations

This section describes operations supported by the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines. The following notations are used here:

A is a sparse matrix;
 B and C are dense matrices;
 D is a diagonal scaling matrix;
 x and y are dense vectors;
 α and β are scalars;

$\text{op}(A)$ is one of the possible operations:

$\text{op}(A) = A$;
 $\text{op}(A) = A^T$ - transpose of A ;
 $\text{op}(A) = A^H$ - conjugated transpose of A .

$\text{inv}(\text{op}(A))$ denotes the inverse of $\text{op}(A)$.

The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines support the following operations:

- computing the vector product between a sparse matrix and a dense vector:

```
y := alpha*op(A)*x + beta*y
```

- solving a single triangular system:

```
y := alpha*inv(op(A))*x
```

- computing a product between sparse matrix and dense matrix:

```
C := alpha*op(A)*B + beta*C
```

- solving a sparse triangular system with multiple right-hand sides:

```
C := alpha*inv(op(A))*B
```

Intel® oneAPI Math Kernel Library (oneMKL) provides an additional set of the Sparse BLAS Level 2 routines with *simplified interfaces*. Each of these routines operates on a matrix of the fixed type. The following operations are supported:

- computing the vector product between a sparse matrix and a dense vector (for general and symmetric matrices):

```
y := op(A)*x
```

- solving a single triangular system (for triangular matrices):

```
y := inv(op(A))*x
```

Matrix type is indicated by the field `<mtype>` in the routine name (see section [Naming Conventions in Sparse BLAS Level 2 and Level 3](#)).

NOTE

The routines with simplified interfaces support only four sparse matrix storage formats, specifically:

CSR format in the 3-array variation accepted in the direct sparse solvers and in the CXML;
 diagonal format accepted in the CXML;
 coordinate format;
 BSR format in the 3-array variation.

Note that routines with both typical (conventional) and simplified interfaces use the same computational kernels that work with certain internal data structures.

The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines do not support in-place operations.

Complete list of all routines is given in the ["Sparse BLAS Level 2 and Level 3 Routines"](#).

Interface Consideration

One-Based and Zero-Based Indexing

The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines support one-based and zero-based indexing of data arrays.

Routines with typical interfaces support zero-based indexing for the following sparse data storage formats: CSR, CSC, BSR, and COO. Routines with simplified interfaces support zero based indexing for the following sparse data storage formats: CSR, BSR, and COO. See the complete list of [Sparse BLAS Level 2 and Level 3 Routines](#).

The one-based indexing uses the convention of starting array indices at 1. The zero-based indexing uses the convention of starting array indices at 0. For example, indices of the 5-element array x can be presented in case of one-based indexing as follows:

Element index: 1 2 3 4 5

Element value: 1.0 5.0 7.0 8.0 9.0

and in case of zero-based indexing as follows:

Element index: 0 1 2 3 4

Element value: 1.0 5.0 7.0 8.0 9.0

The detailed descriptions of the one-based and zero-based variants of the sparse data storage formats are given in the ["Sparse Matrix Storage Formats"](#) in the Appendix "Linear Solvers Basics".

Most parameters of the routines are identical for both one-based and zero-based indexing, but some of them have certain differences. The following table lists all these differences.

Parameter	One-based Indexing	Zero-based Indexing
<i>val</i>	Array containing non-zero elements of the matrix A , its length is $.pntrb[m] - pntrb[1]$	Array containing non-zero elements of the matrix A , its length is $.pntrb[m-1] - pntrb[0]$
<i>pntrb</i>	Array of length m . This array contains row indices, such that $pntrb[i] - pntrb[1]+1$ is the first index of row i in the arrays <i>val</i> and <i>indx</i>	Array of length m . This array contains row indices, such that $pntrb[i] - pntrb[0]$ is the first index of row i in the arrays <i>val</i> and <i>indx</i> .
<i>pntrb</i>	Array of length m . This array contains row indices, such that $pntrb[i] - pntrb[1]$ is the last index of row i in the arrays <i>val</i> and <i>indx</i> .	Array of length m . This array contains row indices, such that $pntrb[i] - pntrb[0]-1$ is the last index of row i in the arrays <i>val</i> and <i>indx</i> .
<i>ia</i>	Array of length $m + 1$, containing indices of elements in the array a , such that $ia[i]$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia[m + 1]$ is equal to the number of non-zeros plus one.	Array of length $m+1$, containing indices of elements in the array a , such that $ia[i]$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia[m]$ is equal to the number of non-zeros.

Parameter	One-based Indexing	Zero-based Indexing
<i>ldb</i>	Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program.	Specifies the second dimension of <i>b</i> as declared in the calling (sub)program.
<i>ldc</i>	Specifies the leading dimension of <i>c</i> as declared in the calling (sub)program.	Specifies the second dimension of <i>c</i> as declared in the calling (sub)program.

Differences Between Intel MKL and NIST* Interfaces

The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 3 routines have the following conventional interfaces:

`mkl_xyyyymm(transa, m, n, k, alpha, matdescra, arg(A), b, ldb, beta, c, ldc)`, for matrix-matrix product;

`mkl_xyyysm(transa, m, n, alpha, matdescra, arg(A), b, ldb, c, ldc)`, for triangular solvers with multiple right-hand sides.

Here *x* denotes data type, and *yyy* - sparse matrix data structure (storage format).

The analogous NIST* Sparse BLAS (NSB) library routines have the following interfaces:

`xyyyymm(transa, m, n, k, alpha, descra, arg(A), b, ldb, beta, c, ldc, work, lwork)`, for matrix-matrix product;

`xyyyysm(transa, m, n, unitd, dv, alpha, descra, arg(A), b, ldb, beta, c, ldc, work, lwork)`, for triangular solvers with multiple right-hand sides.

Some similar arguments are used in both libraries. The argument *transa* indicates what operation is performed and is slightly different in the NSB library (see [Table "Parameter transa"](#)). The arguments *m* and *k* are the number of rows and column in the matrix *A*, respectively, *n* is the number of columns in the matrix *C*. The arguments *alpha* and *beta* are scalar *alpha* and *beta* respectively (*beta* is not used in the Intel® oneAPI Math Kernel Library (oneMKL) triangular solvers.) The arguments *b* and *c* are rectangular arrays with the leading dimension *ldb* and *ldc*, respectively. *arg(A)* denotes the list of arguments that describe the sparse representation of *A*.

Parameter *transa*

	MKL interface	NSB interface	Operation
data type	char *	INTEGER	
value	N or n	0	$\text{op}(A) = A$
	T or t	1	$\text{op}(A) = A^T$
	C or c	2	$\text{op}(A) = A^T$ or $\text{op}(A) = A^H$

Parameter *matdescra*

The parameter *matdescra* describes the relevant characteristic of the matrix *A*. This manual describes *matdescra* as an array of six elements in line with the NIST* implementation. However, only the first four elements of the array are used in the current versions of the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines. Elements *matdescra*[4] and *matdescra*[5] are reserved for future use. Note that whether *matdescra* is described in your application as an array of length 6 or 4 is of no importance because the array is declared as a pointer in the Intel® oneAPI Math Kernel Library (oneMKL) routines. To learn more about declaration of the *matdescra* array, see the Sparse BLAS examples located in the Intel® oneAPI Math

Kernel Library (oneMKL) installation directory: `examples/spblas/` for C. The table below lists elements of the parameter `matdescra`, their Fortran values, and their meanings. The parameter `matdescra` corresponds to the argument `descra` from NSB library.

Possible Values of the Parameter `matdescra` [`descra - 1`]

	MKL interface		NSB interface	Matrix characteristics
	one-based indexing	zero-based indexing		
data type	char *	char *	int *	
1st element	<code>matdescra[1]</code>	<code>matdescra[0]</code>	<code>descra[0]</code>	matrix structure
value	G	G	0	general
	S	S	1	symmetric ($A = A^T$)
	H	H	2	Hermitian ($A = (A^H)$)
	T	T	3	triangular
	A	A	4	skew(anti)-symmetric ($A = -A^T$)
	D	D	5	diagonal
2nd element	<code>matdescra[2]</code>	<code>matdescra[1]</code>	<code>descra[1]</code>	upper/lower triangular indicator
value	L	L	1	lower
	U	U	2	upper
3rd element	<code>matdescra[3]</code>	<code>matdescra[2]</code>	<code>descra[2]</code>	main diagonal type
value	N	N	0	non-unit
	U	U	1	unit
4th element	<code>matdescra[4]</code>	<code>matdescra[3]</code>	<code>descra[3]</code>	type of indexing
value	F		1	one-based indexing
		C	0	zero-based indexing

In some cases possible element values of the parameter `matdescra` depend on the values of other elements. The [Table "Possible Combinations of Element Values of the Parameter `matdescra`"](#) lists all possible combinations of element values for both multiplication routines and triangular solvers.

Possible Combinations of Element Values of the Parameter `matdescra`

Routines	<code>matdescra[0]</code>	<code>matdescra[1]</code>	<code>matdescra[2]</code>	<code>matdescra[3]</code>
Multiplication Routines	G	ignored	ignored	F (default) or C
	S or H	L (default)	N (default)	F (default) or C
	S or H	L (default)	U	F (default) or C
	S or H	U	N (default)	F (default) or C
	S or H	U	U	F (default) or C
	A	L (default)	ignored	F (default) or C

Routines	matdescra[0]	matdescra[1]	matdescra[2]	matdescra[3]
Multiplication Routines and Triangular Solvers	A	U	ignored	F (default) or C
	T	L	U	F (default) or C
	T	L	N	F (default) or C
	T	U	U	F (default) or C
	T	U	N	F (default) or C
	D	ignored	N (default)	F (default) or C
	D	ignored	U	F (default) or C

For a matrix in the skyline format with the main diagonal declared to be a unit, diagonal elements must be stored in the sparse representation even if they are zero. In all other formats, diagonal elements can be stored (if needed) in the sparse representation if they are not zero.

Operations with Partial Matrices

One of the distinctive feature of the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines is a possibility to perform operations only on partial matrices composed of certain parts (triangles and the main diagonal) of the input sparse matrix. It can be done by setting properly first three elements of the parameter *matdescra*.

An arbitrary sparse matrix *A* can be decomposed as

$$A = L + D + U$$

where *L* is the strict lower triangle of *A*, *U* is the strict upper triangle of *A*, *D* is the main diagonal.

Table "Output Matrices for Multiplication Routines" shows correspondence between the output matrices and values of the parameter *matdescra* for the sparse matrix *A* for multiplication routines.

Output Matrices for Multiplication Routines

matdescra[0]	matdescra[1]	matdescra[2]	Output Matrix
G	ignored	ignored	$\alpha * \text{op}(A) * x + \beta * y$ $\alpha * \text{op}(A) * B + \beta * C$
S or H	L	N	$\alpha * \text{op}(L+D+L') * x + \beta * y$ $\alpha * \text{op}(L+D+L') * B + \beta * C$
S or H	L	U	$\alpha * \text{op}(L+I+L') * x + \beta * y$ $\alpha * \text{op}(L+I+L') * B + \beta * C$
S or H	U	N	$\alpha * \text{op}(U'+D+U) * x + \beta * y$ $\alpha * \text{op}(U'+D+U) * B + \beta * C$
S or H	U	U	$\alpha * \text{op}(U'+I+U) * x + \beta * y$ $\alpha * \text{op}(U'+I+U) * B + \beta * C$
T	L	U	$\alpha * \text{op}(L+I) * x + \beta * y$ $\alpha * \text{op}(L+I) * B + \beta * C$
T	L	N	$\alpha * \text{op}(L+D) * x + \beta * y$ $\alpha * \text{op}(L+D) * B + \beta * C$

matdescra[0]	matdescra[1]	matdescra[2]	Output Matrix
T	U	U	$\alpha * \text{op}(U+I) * x + \beta * y$ $\alpha * \text{op}(U+I) * B + \beta * C$
T	U	N	$\alpha * \text{op}(U+D) * x + \beta * y$ $\alpha * \text{op}(U+D) * B + \beta * C$
A	L	ignored	$\alpha * \text{op}(L-L') * x + \beta * y$ $\alpha * \text{op}(L-L') * B + \beta * C$
A	U	ignored	$\alpha * \text{op}(U-U') * x + \beta * y$ $\alpha * \text{op}(U-U') * B + \beta * C$
D	ignored	N	$\alpha * D * x + \beta * y$ $\alpha * D * B + \beta * C$
D	ignored	U	$\alpha * x + \beta * y$ $\alpha * B + \beta * C$

Table "Output Matrices for Triangular Solvers" shows correspondence between the output matrices and values of the parameter *matdescra* for the sparse matrix *A* for triangular solvers.

Output Matrices for Triangular Solvers

matdescra[0]	matdescra[1]	matdescra[2]	Output Matrix
T	L	N	$\alpha * \text{inv}(\text{op}(L)) * x$ $\alpha * \text{inv}(\text{op}(L)) * B$
T	L	U	$\alpha * \text{inv}(\text{op}(L)) * x$ $\alpha * \text{inv}(\text{op}(L)) * B$
T	U	N	$\alpha * \text{inv}(\text{op}(U)) * x$ $\alpha * \text{inv}(\text{op}(U)) * B$
T	U	U	$\alpha * \text{inv}(\text{op}(U)) * x$ $\alpha * \text{inv}(\text{op}(U)) * B$
D	ignored	N	$\alpha * \text{inv}(D) * x$ $\alpha * \text{inv}(D) * B$
D	ignored	U	$\alpha * x$ $\alpha * B$

Sparse BLAS Level 2 and Level 3 Routines.

NOTE The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3 routines are deprecated. Use the corresponding routine from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface as indicated in the description for each routine.

Table "Sparse BLAS Level 2 and Level 3 Routines" lists the sparse BLAS Level 2 and Level 3 routines described in more detail later in this section.

Sparse BLAS Level 2 and Level 3 Routines

Routine/Function	Description
Simplified interface, one-based indexing	
<code>mkl_?csrgemv</code>	Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation)
<code>mkl_?bsrgemv</code>	Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation).
<code>mkl_?coogemv</code>	Computes matrix - vector product of a sparse general matrix in the coordinate format.
<code>mkl_?diagemv</code>	Computes matrix - vector product of a sparse general matrix in the diagonal format.
<code>mkl_?csrsvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation)
<code>mkl_?bsrsvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation).
<code>mkl_?coosvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format.
<code>mkl_?diasvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the diagonal format.
<code>mkl_?csrtrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation).
<code>mkl_?bsrtrsv</code>	Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation).
<code>mkl_?cootrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the coordinate format.
<code>mkl_?diatrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the diagonal format.
Simplified interface, zero-based indexing	
<code>mkl_cspblas_?csrgemv</code>	Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?bsrgemv</code>	Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?coogemv</code>	Computes matrix - vector product of a sparse general matrix in the coordinate format with zero-based indexing.
<code>mkl_cspblas_?csrsvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation) with zero-based indexing
<code>mkl_cspblas_?bsrsvmv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation) with zero-based indexing.

Routine/Function	Description
<code>mkl_cspblas_?coosymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format with zero-based indexing.
<code>mkl_cspblas_?csrtrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?bsrtrsv</code>	Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?cootrsv</code>	Triangular solver with simplified interface for a sparse matrix in the coordinate format with zero-based indexing.

Typical (conventional) interface, one-based and zero-based indexing

<code>mkl_?csrmmv</code>	Computes matrix - vector product of a sparse matrix in the CSR format.
<code>mkl_?bsrmmv</code>	Computes matrix - vector product of a sparse matrix in the BSR format.
<code>mkl_?cscmmv</code>	Computes matrix - vector product for a sparse matrix in the CSC format.
<code>mkl_?coomv</code>	Computes matrix - vector product for a sparse matrix in the coordinate format.
<code>mkl_?csrsv</code>	Solves a system of linear equations for a sparse matrix in the CSR format.
<code>mkl_?bsrsv</code>	Solves a system of linear equations for a sparse matrix in the BSR format.
<code>mkl_?cscsv</code>	Solves a system of linear equations for a sparse matrix in the CSC format.
<code>mkl_?coosv</code>	Solves a system of linear equations for a sparse matrix in the coordinate format.
<code>mkl_?csrmm</code>	Computes matrix - matrix product of a sparse matrix in the CSR format
<code>mkl_?bsrmm</code>	Computes matrix - matrix product of a sparse matrix in the BSR format.
<code>mkl_?cscmm</code>	Computes matrix - matrix product of a sparse matrix in the CSC format
<code>mkl_?coomm</code>	Computes matrix - matrix product of a sparse matrix in the coordinate format.
<code>mkl_?csrsm</code>	Solves a system of linear matrix equations for a sparse matrix in the CSR format.
<code>mkl_?bsrsm</code>	Solves a system of linear matrix equations for a sparse matrix in the BSR format.
<code>mkl_?cscsm</code>	Solves a system of linear matrix equations for a sparse matrix in the CSC format.

Routine/Function	Description
<code>mkl_?coosm</code>	Solves a system of linear matrix equations for a sparse matrix in the coordinate format.
Typical (conventional) interface, one-based indexing	
<code>mkl_?diamv</code>	Computes matrix - vector product of a sparse matrix in the diagonal format.
<code>mkl_?skymv</code>	Computes matrix - vector product for a sparse matrix in the skyline storage format.
<code>mkl_?diasv</code>	Solves a system of linear equations for a sparse matrix in the diagonal format.
<code>mkl_?skysv</code>	Solves a system of linear equations for a sparse matrix in the skyline format.
<code>mkl_?diamm</code>	Computes matrix - matrix product of a sparse matrix in the diagonal format.
<code>mkl_?skymm</code>	Computes matrix - matrix product of a sparse matrix in the skyline storage format.
<code>mkl_?diasm</code>	Solves a system of linear matrix equations for a sparse matrix in the diagonal format.
<code>mkl_?skysm</code>	Solves a system of linear matrix equations for a sparse matrix in the skyline storage format.
Auxiliary routines	
Matrix converters	
<code>mkl_?dnscsr</code>	Converts a sparse matrix in uncompressed representation to CSR format (3-array variation) and vice versa.
<code>mkl_?csrcoo</code>	Converts a sparse matrix in CSR format (3-array variation) to coordinate format and vice versa.
<code>mkl_?csrbsr</code>	Converts a sparse matrix in CSR format to BSR format (3-array variations) and vice versa.
<code>mkl_?csrcsc</code>	Converts a sparse matrix in CSR format to CSC format and vice versa (3-array variations).
<code>mkl_?csrdia</code>	Converts a sparse matrix in CSR format (3-array variation) to diagonal format and vice versa.
<code>mkl_?csrsky</code>	Converts a sparse matrix in CSR format (3-array variation) to sky line format and vice versa.
Operations on sparse matrices	
<code>mkl_?csradd</code>	Computes the sum of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.
<code>mkl_?csrmultcsr</code>	Computes the product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.

Routine/Function	Description
<code>mkl_?csrmultd</code>	Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing. The result is stored in the dense matrix.

mkl_?csrgemv

Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
void mkl_scsrgemv (const char *transa , const MKL_INT *m , const float *a , const
MKL_INT *ia , const MKL_INT *ja , const float *x , float *y );

void mkl_dcsrgemv (const char *transa , const MKL_INT *m , const double *a , const
MKL_INT *ia , const MKL_INT *ja , const double *x , double *y );

void mkl_ccsrgemv (const char *transa , const MKL_INT *m , const MKL_Complex8 *a ,
const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_zcsrgemv (const char *transa , const MKL_INT *m , const MKL_Complex16 *a ,
const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- `mkl.h`

Description

This routine is deprecated. Use `mkl_sparse_?_mv` from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrgemv` routine performs a matrix-vector operation defined as

$$y := A * x$$

or

$$y := A^T * x,$$

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the CSR format (3-array variation), A^T is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<code>transa</code>	Specifies the operation. If <code>transa = 'N'</code> or <code>'n'</code> , then as $y := A * x$ If <code>transa = 'T'</code> or <code>'t'</code> or <code>'C'</code> or <code>'c'</code> , then $y := A^T * x$,
---------------------	---

<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>a</i>	Array containing non-zero elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	Array of length $m + 1$, containing indices of elements in the array <i>a</i> , such that $ia[i] - ia[0]$ is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element $ia[m] - ia[0]$ is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	Array containing the column indices plus one for each non-zero element of the matrix <i>A</i> . Its length is equal to the length of the array <i>a</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>x</i>	Array, size is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	Array, size at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	---

mkl_?bsrgemv

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
void mkl_sbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb , const
float *a , const MKL_INT *ia , const MKL_INT *ja , const float *x , float *y );

void mkl_dbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb , const
double *a , const MKL_INT *ia , const MKL_INT *ja , const double *x , double *y );

void mkl_cbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb , const
MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x ,
MKL_Complex8 *y );

void mkl_zbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb , const
MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex16 *x ,
MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?bsrgemv` routine performs a matrix-vector operation defined as

$$y := A * x$$

or

$$y := A^T * x,$$

where:

x and y are vectors,

A is an m -by- m block sparse square matrix in the BSR format (3-array variation), A^T is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A * x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A^T * x$,
<i>m</i>	Number of block rows of the matrix A .
<i>lb</i>	Size of the block in the matrix A .
<i>a</i>	Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to <i>values</i> array description in BSR Format for more details.
<i>ia</i>	Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that $ia[i] - ia[0]$ is the index in the array <i>a</i> of the first non-zero element from the row i . The value of the last element $ia[m] - ia[0]$ is equal to the number of non-zero blocks. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	Array containing the column indices plus one for each non-zero block in the matrix A . Its length is equal to the number of non-zero blocks of the matrix A . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	Array, size $(m * lb)$. On entry, the array <i>x</i> must contain the vector x .

Output Parameters

<i>y</i>	Array, size at least $(m * lb)$. On exit, the array <i>y</i> must contain the vector y .
----------	--

mkl_?coogemv

Computes matrix-vector product of a sparse general matrix stored in the coordinate format with one-based indexing (deprecated).

Syntax

```
void mkl_scoogemv (const char *transa , const MKL_INT *m , const float *val , const
MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const float *x , float
*y );

void mkl_dcoogemv (const char *transa , const MKL_INT *m , const double *val , const
MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const double *x , double
*y );

void mkl_ccoogemv (const char *transa , const MKL_INT *m , const MKL_Complex8 *val ,
const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex8
*x , MKL_Complex8 *y );

void mkl_zcoogemv (const char *transa , const MKL_INT *m , const MKL_Complex16 *val ,
const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const
MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?coogemv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := AT*x,
```

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the coordinate format, A^T is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

transa

Specifies the operation.

If *transa* = 'N' or 'n', then the matrix-vector product is computed as
 $y := A*x$

If *transa* = 'T' or 't' or 'C' or 'c', then the matrix-vector product is
 computed as $y := A^T*x$,

<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>val</i>	Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	Array of length <i>nnz</i> , contains the row indices plus one for each non-zero element of the matrix <i>A</i> . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	Array of length <i>nnz</i> , contains the column indices plus one for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	Specifies the number of non-zero element of the matrix <i>A</i> . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	Array, size is <i>m</i> . One entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	Array, size at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	---

mkl_?diagemv

Computes matrix - vector product of a sparse general matrix stored in the diagonal format with one-based indexing (deprecated).

Syntax

```
void mkl_sdiagemv (const char *transa , const MKL_INT *m , const float *val , const
MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const float *x , float
*y );

void mkl_ddiagemv (const char *transa , const MKL_INT *m , const double *val , const
MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const double *x , double
*y );

void mkl_cdiagemv (const char *transa , const MKL_INT *m , const MKL_Complex8 *val ,
const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex8
*x , MKL_Complex8 *y );

void mkl_zdiagemv (const char *transa , const MKL_INT *m , const MKL_Complex16 *val ,
const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex16
*x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use `mkl_sparse_?_mv` from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?diagmv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := AT*x,
```

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the diagonal storage format, A^T is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := A*x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := A^T*x$,
<i>m</i>	Number of rows of the matrix A .
<i>val</i>	Two-dimensional array of size $lval*ndiag$, contains non-zero diagonals of the matrix A . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix A . Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	Specifies the number of non-zero diagonals of the matrix A .
<i>x</i>	Array, size is m . On entry, the array <i>x</i> must contain the vector x .

Output Parameters

<i>y</i>	Array, size at least m . On exit, the array <i>y</i> must contain the vector y .
----------	---

`mkl_?csrsvmv`

Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
void mkl_scsrsymv (const char *uplo , const MKL_INT *m , const float *a , const MKL_INT
*ia , const MKL_INT *ja , const float *x , float *y );
```

```
void mkl_dcsrsymv (const char *uplo , const MKL_INT *m , const double *a , const
MKL_INT *ia , const MKL_INT *ja , const double *x , double *y );
```

```
void mkl_ccsrsymv (const char *uplo , const MKL_INT *m , const MKL_Complex8 *a , const
MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x , MKL_Complex8 *y );
```

```
void mkl_zcsrsymv (const char *uplo , const MKL_INT *m , const MKL_Complex16 *a , const
MKL_INT *ia , const MKL_INT *ja , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrsymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation).

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>m</i>	Number of rows of the matrix A .
<i>a</i>	Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	Array of length $m + 1$, containing indices of elements in the array a , such that $ia[i] - ia[0]$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia[m] - ia[0]$ is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.

<i>ja</i>	Array containing the column indices plus one for each non-zero element of the matrix <i>A</i> . Its length is equal to the length of the array <i>a</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>x</i>	Array, size is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	Array, size at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	---

mkl_?bsrsymv

Computes matrix-vector product of a sparse symmetrical matrix stored in the BSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
void mkl_sbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb , const
float *a , const MKL_INT *ia , const MKL_INT *ja , const float *x , float *y );
void mkl_dbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb , const
double *a , const MKL_INT *ia , const MKL_INT *ja , const double *x , double *y );
void mkl_cbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb , const
MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x ,
MKL_Complex8 *y );
void mkl_zbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb , const
MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex16 *x ,
MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?bsrsymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation).

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies whether the upper or low triangle of the matrix <i>A</i> is considered. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>m</i>	Number of block rows of the matrix <i>A</i> .
<i>lb</i>	Size of the block in the matrix <i>A</i> .
<i>a</i>	Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb*lb</i> . Refer to <i>values</i> array description in BSR Format for more details.
<i>ia</i>	Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that <i>ia</i> [<i>i</i>] - <i>ia</i> [0] is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> [<i>m</i>] - <i>ia</i> [0] is equal to the number of non-zero blocks. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	Array containing the column indices plus one for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks of the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	Array, size $(m*lb)$. On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	Array, size at least $(m*lb)$. On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	---

mkl_?coosymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with one-based indexing (deprecated).

Syntax

```
void mkl_scoosymv (const char *uplo , const MKL_INT *m , const float *val , const
MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const float *x , float
*y );

void mkl_dcoosymv (const char *uplo , const MKL_INT *m , const double *val , const
MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const double *x , double
*y );

void mkl_ccoosymv (const char *uplo , const MKL_INT *m , const MKL_Complex8 *val ,
const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex8
*x , MKL_Complex8 *y );
```

```
void mkl_zcoosymv (const char *uplo , const MKL_INT *m , const MKL_Complex16 *val ,
const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const
MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?coosymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>m</i>	Number of rows of the matrix A .
<i>val</i>	Array of length <i>nnz</i> , contains non-zero elements of the matrix A in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	Array of length <i>nnz</i> , contains the row indices plus one for each non-zero element of the matrix A . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	Array of length <i>nnz</i> , contains the column indices plus one for each non-zero element of the matrix A . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	Specifies the number of non-zero element of the matrix A . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	Array, size is <i>m</i> . On entry, the array <i>x</i> must contain the vector x .

Output Parameters

y Array, size at least *m*.
On exit, the array *y* must contain the vector *y*.

mkl_?diasymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the diagonal format with one-based indexing (deprecated).

Syntax

```
void mkl_sdiasymv (const char *uplo , const MKL_INT *m , const float *val , const
MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const float *x , float
*y );

void mkl_ddiasymv (const char *uplo , const MKL_INT *m , const double *val , const
MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const double *x , double
*y );

void mkl_cdiasymv (const char *uplo , const MKL_INT *m , const MKL_Complex8 *val ,
const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex8
*x , MKL_Complex8 *y );

void mkl_zdiasymv (const char *uplo , const MKL_INT *m , const MKL_Complex16 *val ,
const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex16
*x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?diasymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

uplo Specifies whether the upper or low triangle of the matrix *A* is used.
If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

	If <code>uplo = 'L'</code> or <code>'l'</code> , then the low triangle of the matrix <i>A</i> is used.
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>val</i>	Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	Leading dimension of <i>val</i> , <i>lval</i> ≥ <i>m</i> . Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> . Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>x</i>	Array, size is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	Array, size at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	---

mkl_?csrtrsv

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
void mkl_scsrtrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const float *a , const MKL_INT *ia , const MKL_INT *ja , const float *x ,
float *y );

void mkl_dcsrtrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const double *a , const MKL_INT *ia , const MKL_INT *ja , const double
*x , double *y );

void mkl_ccsrtrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const
MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_zcsrtrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const
MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- `mkl.h`

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3 array variation):

$$A^*y = x$$

or

$$A^T*y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A^*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A^T*y = x$,
<i>diag</i>	Specifies whether A is unit triangular. If <i>diag</i> = 'U' or 'u', then A is a unit triangular. If <i>diag</i> = 'N' or 'n', then A is not unit triangular.
<i>m</i>	Number of rows of the matrix A .
<i>a</i>	Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

ia Array of length $m + 1$, containing indices of elements in the array *a*, such that $ia[i] - ia[0]$ is the index in the array *a* of the first non-zero element from the row *i*. The value of the last element $ia[m] - ia[0]$ is equal to the number of non-zeros. Refer to [rowIndex](#) array description in [Sparse Matrix Storage Formats](#) for more details.

ja Array containing the column indices plus one for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *a*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

NOTE

Column indices must be sorted in increasing order for each row.

x Array, size is *m*.
On entry, the array *x* must contain the vector *x*.

Output Parameters

y Array, size at least *m*.
Contains the vector *y*.

mkl_?bsrtrsv

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
void mkl_sbsrtrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const MKL_INT *lb , const float *a , const MKL_INT *ia , const MKL_INT
*ja , const float *x , float *y );

void mkl_dbsrtrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const MKL_INT *lb , const double *a , const MKL_INT *ia , const MKL_INT
*ja , const double *x , double *y );

void mkl_cbsrtrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const MKL_INT *lb , const MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_zbsrtrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const MKL_INT *lb , const MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- `mkl.h`

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?bsrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) :

```
y := A*x
```

or

```
y := AT*x,
```

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A^T*x$.
<i>diag</i>	Specifies whether A is a unit triangular matrix. If <i>diag</i> = 'U' or 'u', then A is a unit triangular. If <i>diag</i> = 'N' or 'n', then A is not a unit triangular.
<i>m</i>	Number of block rows of the matrix A .
<i>lb</i>	Size of the block in the matrix A .
<i>a</i>	Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb*lb$. Refer to <i>values</i> array description in BSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>ia</i>	Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that $ia[I] - ia[0]$ is the index in the array <i>a</i> of the first non-zero element from the row <i>I</i> . The value of the last element $ia[m] - ia[0]$ is equal to the number of non-zero blocks. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	Array containing the column indices plus one for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks of the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	Array, size $(m \times lb)$. On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	Array, size at least $(m \times lb)$. On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	--

mkl_?cootrsv

Triangular solvers with simplified interface for a sparse matrix in the coordinate format with one-based indexing (deprecated).

Syntax

```
void mkl_scootrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const float *val , const MKL_INT *rowind , const MKL_INT *colind , const
MKL_INT *nnz , const float *x , float *y );

void mkl_dcootrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const double *val , const MKL_INT *rowind , const MKL_INT *colind , const
MKL_INT *nnz , const double *x , double *y );

void mkl_ccootrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const MKL_Complex8 *val , const MKL_INT *rowind , const MKL_INT *colind ,
const MKL_INT *nnz , const MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_zcootrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const MKL_Complex16 *val , const MKL_INT *rowind , const MKL_INT *colind ,
const MKL_INT *nnz , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?cootrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format:

$$A \cdot y = x$$

or

$$A^T * y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies whether the upper or low triangle of the matrix A is considered. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A * y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A^T * y = x$,
<i>diag</i>	Specifies whether A is unit triangular. If <i>diag</i> = 'U' or 'u', then A is unit triangular. If <i>diag</i> = 'N' or 'n', then A is not unit triangular.
<i>m</i>	Number of rows of the matrix A .
<i>val</i>	Array of length <i>nnz</i> , contains non-zero elements of the matrix A in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	Array of length <i>nnz</i> , contains the row indices plus one for each non-zero element of the matrix A . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	Array of length <i>nnz</i> , contains the column indices plus one for each non-zero element of the matrix A . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	Specifies the number of non-zero element of the matrix A . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	Array, size is <i>m</i> . On entry, the array <i>x</i> must contain the vector x .

Output Parameters

<i>y</i>	Array, size at least <i>m</i> . Contains the vector y .
----------	--

mkl_?diatrsv

Triangular solvers with simplified interface for a sparse matrix in the diagonal format with one-based indexing (deprecated).

Syntax

```
void mkl_sdiatrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const float *val , const MKL_INT *lval , const MKL_INT *idiag , const
MKL_INT *ndiag , const float *x , float *y );

void mkl_ddiatrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const double *val , const MKL_INT *lval , const MKL_INT *idiag , const
MKL_INT *ndiag , const double *x , double *y );

void mkl_cdiatrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const MKL_Complex8 *val , const MKL_INT *lval , const MKL_INT *idiag ,
const MKL_INT *ndiag , const MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_zdiatrsv (const char *uplo , const char *transa , const char *diag , const
MKL_INT *m , const MKL_Complex16 *val , const MKL_INT *lval , const MKL_INT *idiag ,
const MKL_INT *ndiag , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?diatrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

$$A*y = x$$

or

$$A^T*y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	Specifies the system of linear equations.

	If <i>transa</i> = 'N' or 'n', then $A^*y = x$
	If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A^T*y = x$,
<i>diag</i>	Specifies whether <i>A</i> is unit triangular. If <i>diag</i> = 'U' or 'u', then <i>A</i> is unit triangular. If <i>diag</i> = 'N' or 'n', then <i>A</i> is not unit triangular.
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>val</i>	Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> .

NOTE

All elements of this array must be sorted in increasing order.

Refer to *distance* array description in [Diagonal Storage Scheme](#) for more details.

<i>ndiag</i>	Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>x</i>	Array, size is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	Array, size at least <i>m</i> . Contains the vector <i>y</i> .
----------	---

[mkl_cspblas_csrgemv](#)

Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with zero-based indexing (deprecated).

Syntax

```
void mkl_cspblas_scsrgemv (const char *transa , const MKL_INT *m , const float *a ,
const MKL_INT *ia , const MKL_INT *ja , const float *x , float *y );

void mkl_cspblas_dcsrgemv (const char *transa , const MKL_INT *m , const double *a ,
const MKL_INT *ia , const MKL_INT *ja , const double *x , double *y );

void mkl_cspblas_ccsrgemv (const char *transa , const MKL_INT *m , const MKL_Complex8
*a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_cspblas_zcsrgemv (const char *transa , const MKL_INT *m , const MKL_Complex16
*a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex16 *x , MKL_Complex16
*y );
```

Include Files

- `mkl.h`

Description

This routine is deprecated. Use `mkl_sparse_?_mv` from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_cspblas_?csrgermv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := AT*x,
```

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the CSR format (3-array variation) with zero-based indexing, A^T is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

<i>transa</i>	<p>Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A^T*x$,</p>
<i>m</i>	Number of rows of the matrix A .
<i>a</i>	Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	Array of length $m + 1$, containing indices of elements in the array <i>a</i> , such that <i>ia</i> [<i>I</i>] is the index in the array <i>a</i> of the first non-zero element from the row <i>I</i> . The value of the last element <i>ia</i> [<i>m</i>] is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	<p>Array containing the column indices for each non-zero element of the matrix A.</p> <p>Its length is equal to the length of the array <i>a</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>x</i>	<p>Array, size is <i>m</i>.</p> <p>One entry, the array <i>x</i> must contain the vector x.</p>

Output Parameters

y

Array, size at least m .

On exit, the array y must contain the vector y .

mkl_cspblas_?bsrgemv

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with zero-based indexing (deprecated).

Syntax

```
void mkl_cspblas_sbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb ,
const float *a , const MKL_INT *ia , const MKL_INT *ja , const float *x , float *y );
void mkl_cspblas_dbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb ,
const double *a , const MKL_INT *ia , const MKL_INT *ja , const double *x , double
*y );
void mkl_cspblas_cbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb ,
const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x ,
MKL_Complex8 *y );
void mkl_cspblas_zbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb ,
const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex16
*x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_cspblas_?bsrgemv` routine performs a matrix-vector operation defined as

$$y := A * x$$

or

$$y := A^T * x,$$

where:

x and y are vectors,

A is an m -by- m block sparse square matrix in the BSR format (3-array variation) with zero-based indexing, A^T is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

<i>transa</i>	<p>Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A \cdot x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A^T \cdot x$,</p>
<i>m</i>	Number of block rows of the matrix <i>A</i> .
<i>lb</i>	Size of the block in the matrix <i>A</i> .
<i>a</i>	<p>Array containing elements of non-zero blocks of the matrix <i>A</i>. Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb</i>*<i>lb</i>. Refer to <i>values</i> array description in BSR Format for more details.</p>
<i>ia</i>	<p>Array of length $(m + 1)$, containing indices of block in the array <i>a</i>, such that <i>ia</i>[<i>i</i>] is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i>. The value of the last element <i>ia</i>[<i>m</i>] is equal to the number of non-zero blocks. Refer to <i>rowIndex</i> array description in BSR Format for more details.</p>
<i>ja</i>	<p>Array containing the column indices for each non-zero block in the matrix <i>A</i>. Its length is equal to the number of non-zero blocks of the matrix <i>A</i>. Refer to <i>columns</i> array description in BSR Format for more details.</p>
<i>x</i>	<p>Array, size $(m \cdot lb)$.</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>.</p>

Output Parameters

<i>y</i>	<p>Array, size at least $(m \cdot lb)$.</p> <p>On exit, the array <i>y</i> must contain the vector <i>y</i>.</p>
----------	---

mkl_cspblas_?coogemv

Computes matrix - vector product of a sparse general matrix stored in the coordinate format with zero-based indexing (deprecated).

Syntax

```
void mkl_cspblas_scoogemv (const char *transa , const MKL_INT *m , const float *val ,
const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const float *x ,
float *y );
```

```
void mkl_cspblas_dcoogemv (const char *transa , const MKL_INT *m , const double *val ,
const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const double *x ,
double *y );
```

```
void mkl_cspblas_ccoogemv (const char *transa , const MKL_INT *m , const MKL_Complex8
*val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const
MKL_Complex8 *x , MKL_Complex8 *y );
```

```
void mkl_cspblas_zcoogemv (const char *transa , const MKL_INT *m , const MKL_Complex16
*val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const
MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- `mkl.h`

Description

This routine is deprecated. Use `mkl_sparse_?_mv` from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_cspblas_dcoogemv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := AT*x,
```

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the coordinate format with zero-based indexing, A^T is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

<i>transa</i>	<p>Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A^T*x$.</p>
<i>m</i>	Number of rows of the matrix A .
<i>val</i>	<p>Array of length <i>nnz</i>, contains non-zero elements of the matrix A in the arbitrary order.</p> <p>Refer to <i>values</i> array description in Coordinate Format for more details.</p>
<i>rowind</i>	<p>Array of length <i>nnz</i>, contains the row indices for each non-zero element of the matrix A.</p> <p>Refer to <i>rows</i> array description in Coordinate Format for more details.</p>
<i>colind</i>	<p>Array of length <i>nnz</i>, contains the column indices for each non-zero element of the matrix A. Refer to <i>columns</i> array description in Coordinate Format for more details.</p>
<i>nnz</i>	<p>Specifies the number of non-zero element of the matrix A.</p> <p>Refer to <i>nnz</i> description in Coordinate Format for more details.</p>
<i>x</i>	<p>Array, size is m.</p> <p>On entry, the array x must contain the vector x.</p>

<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>a</i>	Array containing non-zero elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	Array of length $m + 1$, containing indices of elements in the array <i>a</i> , such that <i>ia</i> [<i>i</i>] is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> [<i>m</i>] is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	Array containing the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to the length of the array <i>a</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>x</i>	Array, size is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	Array, size at least <i>m</i> . On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	---

mkl_cspblas_?bsrsymv

Computes matrix-vector product of a sparse symmetrical matrix stored in the BSR format (3-arrays variation) with zero-based indexing (deprecated).

Syntax

```
void mkl_cspblas_sbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb ,
const float *a , const MKL_INT *ia , const MKL_INT *ja , const float *x , float *y );
void mkl_cspblas_dbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb ,
const double *a , const MKL_INT *ia , const MKL_INT *ja , const double *x , double
*y );
void mkl_cspblas_cbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb ,
const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x ,
MKL_Complex8 *y );
void mkl_cspblas_zbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb ,
const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex16
*x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_cspblas_?bsrsymv` routine performs a matrix-vector operation defined as

$$y := A * x$$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation) with zero-based indexing.

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

<code>uplo</code>	Specifies whether the upper or low triangle of the matrix A is used. If <code>uplo</code> = 'U' or 'u', then the upper triangle of the matrix A is used. If <code>uplo</code> = 'L' or 'l', then the low triangle of the matrix A is used.
<code>m</code>	Number of block rows of the matrix A .
<code>lb</code>	Size of the block in the matrix A .
<code>a</code>	Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to <code>values</code> array description in BSR Format for more details.
<code>ia</code>	Array of length $(m + 1)$, containing indices of block in the array a , such that <code>ia[i]</code> is the index in the array a of the first non-zero element from the row i . The value of the last element <code>ia[m]</code> is equal to the number of non-zero blocks. Refer to <code>rowIndex</code> array description in BSR Format for more details.
<code>ja</code>	Array containing the column indices for each non-zero block in the matrix A . Its length is equal to the number of non-zero blocks of the matrix A . Refer to <code>columns</code> array description in BSR Format for more details.
<code>x</code>	Array, size $(m * lb)$. On entry, the array x must contain the vector x .

Output Parameters

<code>y</code>	Array, size at least $(m * lb)$. On exit, the array y must contain the vector y .
----------------	---

`mkl_cspblas_?coosymv`

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with zero-based indexing (deprecated).

Syntax

```
void mkl_cspblas_scoosymv (const char *uplo , const MKL_INT *m , const float *val ,
const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const float *x ,
float *y );

void mkl_cspblas_dcoosymv (const char *uplo , const MKL_INT *m , const double *val ,
const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const double *x ,
double *y );

void mkl_cspblas_ccoosymv (const char *uplo , const MKL_INT *m , const MKL_Complex8
*val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const
MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_cspblas_zcoosymv (const char *uplo , const MKL_INT *m , const MKL_Complex16
*val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const
MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_cspblas_?coosymv` routine performs a matrix-vector operation defined as

$$y := A * x$$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format with zero-based indexing.

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>m</i>	Number of rows of the matrix A .
<i>val</i>	Array of length <i>nnz</i> , contains non-zero elements of the matrix A in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	Array of length <i>nnz</i> , contains the row indices for each non-zero element of the matrix A .

Refer to `rows` array description in [Coordinate Format](#) for more details.

`colind`

Array of length `nnz`, contains the column indices for each non-zero element of the matrix `A`. Refer to `columns` array description in [Coordinate Format](#) for more details.

`nnz`

Specifies the number of non-zero element of the matrix `A`.

Refer to `nnz` description in [Coordinate Format](#) for more details.

`x`

Array, size is `m`.

On entry, the array `x` must contain the vector `x`.

Output Parameters

`y`

Array, size at least `m`.

On exit, the array `y` must contain the vector `y`.

`mkl_cspblas_?csrtrsv`

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing (deprecated).

Syntax

```
void mkl_cspblas_scsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const float *a , const MKL_INT *ia , const MKL_INT *ja , const float
*x , float *y );
```

```
void mkl_cspblas_dcsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const double *a , const MKL_INT *ia , const MKL_INT *ja , const
double *x , double *y );
```

```
void mkl_cspblas_ccsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_Complex8 *x , MKL_Complex8 *y );
```

```
void mkl_cspblas_zcsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- `mkl.h`

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_cspblas_?csrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3-array variation) with zero-based indexing:

$$A * y = x$$

or

$$A^T * y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies whether the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A^T*y = x$,
<i>diag</i>	Specifies whether matrix A is unit triangular. If <i>diag</i> = 'U' or 'u', then A is unit triangular. If <i>diag</i> = 'N' or 'n', then A is not unit triangular.
<i>m</i>	Number of rows of the matrix A .
<i>a</i>	Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>ia</i>	Array of length $m+1$, containing indices of elements in the array <i>a</i> , such that <i>ia</i> [<i>i</i>] is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> [<i>m</i>] is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	Array containing the column indices for each non-zero element of the matrix A . Its length is equal to the length of the array <i>a</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.

NOTE

Column indices must be sorted in increasing order for each row.

x Array, size is m .
On entry, the array x must contain the vector x .

Output Parameters

y Array, size at least m .
Contains the vector y .

mkl_cspblas_?bsrtrsv

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing (deprecated).

Syntax

```
void mkl_cspblas_sbsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_INT *lb , const float *a , const MKL_INT *ia , const
MKL_INT *ja , const float *x , float *y );
```

```
void mkl_cspblas_dbsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_INT *lb , const double *a , const MKL_INT *ia , const
MKL_INT *ja , const double *x , double *y );
```

```
void mkl_cspblas_cbsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_INT *lb , const MKL_Complex8 *a , const MKL_INT *ia ,
const MKL_INT *ja , const MKL_Complex8 *x , MKL_Complex8 *y );
```

```
void mkl_cspblas_zbsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_INT *lb , const MKL_Complex16 *a , const MKL_INT *ia ,
const MKL_INT *ja , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_cspblas_?bsrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing:

$$y := A * x$$

or

$$y := A^T * x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies the upper or low triangle of the matrix A is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := A*x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := A^T*x$.
<i>diag</i>	Specifies whether matrix A is unit triangular or not. If <i>diag</i> = 'U' or 'u', A is unit triangular. If <i>diag</i> = 'N' or 'n', A is not unit triangular.
<i>m</i>	Number of block rows of the matrix A .
<i>lb</i>	Size of the block in the matrix A .
<i>a</i>	Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb*lb$. Refer to <i>values</i> array description in BSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>ia</i>	Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that <i>ia</i> [<i>I</i>] is the index in the array <i>a</i> of the first non-zero element from the row <i>I</i> . The value of the last element <i>ia</i> [<i>m</i>] is equal to the number of non-zero blocks. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	Array containing the column indices for each non-zero block in the matrix A . Its length is equal to the number of non-zero blocks of the matrix A . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	Array, size $(m*lb)$. On entry, the array <i>x</i> must contain the vector x .

Output Parameters

y Array, size at least $(m \cdot lb)$.
On exit, the array y must contain the vector y .

mkl_cspblas_?cootrsv

Triangular solvers with simplified interface for a sparse matrix in the coordinate format with zero-based indexing (deprecated).

Syntax

```
void mkl_cspblas_scootrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const float *val , const MKL_INT *rowind , const MKL_INT *colind ,
const MKL_INT *nnz , const float *x , float *y );

void mkl_cspblas_dcootrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const double *val , const MKL_INT *rowind , const MKL_INT *colind ,
const MKL_INT *nnz , const double *x , double *y );

void mkl_cspblas_ccootrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_Complex8 *val , const MKL_INT *rowind , const MKL_INT
*colind , const MKL_INT *nnz , const MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_cspblas_zcootrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_Complex16 *val , const MKL_INT *rowind , const MKL_INT
*colind , const MKL_INT *nnz , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_cspblas_?cootrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format with zero-based indexing:

$$A * y = x$$

or

$$A^T * y = x,$$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

<i>uplo</i>	Specifies whether the upper or low triangle of the matrix <i>A</i> is considered. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A^T*y = x$,
<i>diag</i>	Specifies whether <i>A</i> is unit triangular. If <i>diag</i> = 'U' or 'u', then <i>A</i> is unit triangular. If <i>diag</i> = 'N' or 'n', then <i>A</i> is not unit triangular.
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>val</i>	Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	Array of length <i>nnz</i> , contains the row indices for each non-zero element of the matrix <i>A</i> . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	Specifies the number of non-zero element of the matrix <i>A</i> . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>x</i>	Array, size is <i>m</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	Array, size at least <i>m</i> . Contains the vector <i>y</i> .
----------	---

mkl_?csrmv

Computes matrix - vector product of a sparse matrix stored in the CSR format (deprecated).

Syntax

```
void mkl_scsr_mv (const char *transa , const MKL_INT *m , const MKL_INT *k , const float
*alpha , const char *matdescra , const float *val , const MKL_INT *indx , const MKL_INT
*pntrb , const MKL_INT *pntrc , const float *x , const float *beta , float *y );

void mkl_dcsr_mv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *indx , const
MKL_INT *pntrb , const MKL_INT *pntrc , const double *x , const double *beta , double
*y );
```

```
void mkl_ccsrmv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*indx , const MKL_INT *pntrb , const MKL_INT *pntre , const MKL_Complex8 *x , const
MKL_Complex8 *beta , MKL_Complex8 *y );

void mkl_zcsrmv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*indx , const MKL_INT *pntrb , const MKL_INT *pntre , const MKL_Complex16 *x , const
MKL_Complex16 *beta , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrmv` routine performs a matrix-vector operation defined as

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

or

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

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in the CSR format, *A*^T is the transpose of *A*.

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha A^T x + \beta y$,
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>k</i>	Number of columns of the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Array containing non-zero elements of the matrix <i>A</i> .

Its length is `pntre[m-1] - pntrb[0]`.

Refer to *values* array description in [CSR Format](#) for more details.

indx

For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix *A*. For zero-based indexing, array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to length of the *val* array.

Refer to *columns* array description in [CSR Format](#) for more details.

pntrb

Array of length *m*.

This array contains row indices, such that `pntrb[i] - pntrb[0]` is the first index of row *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in [CSR Format](#) for more details.

pntre

Array of length *m*.

This array contains row indices, such that `pntre[i] - pntrb[0]-1` is the last index of row *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in [CSR Format](#) for more details.

x

Array, size at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.

beta

Specifies the scalar *beta*.

y

Array, size at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y

Overwritten by the updated vector *y*.

mkl_?bsrmv

Computes matrix - vector product of a sparse matrix stored in the BSR format (deprecated).

Syntax

```
void mkl_sbsrmv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_INT *lb , const float *alpha , const char *matdescra , const float *val , const
MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const float *x , const
float *beta , float *y );
```

```
void mkl_dbsrmv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_INT *lb , const double *alpha , const char *matdescra , const double *val , const
MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const double *x , const
double *beta , double *y );
```

```
void mkl_cbsrmv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_INT *lb , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8
*val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const
MKL_Complex8 *x , const MKL_Complex8 *beta , MKL_Complex8 *y );
```

```
void mkl_zbsrmv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_INT *lb , const MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16
*val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const
MKL_Complex16 *x , const MKL_Complex16 *beta , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?bsrmv` routine performs a matrix-vector operation defined as

$$y := \alpha A x + \beta y$$

or

$$y := \alpha A^T x + \beta y,$$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* block sparse matrix in the BSR format, A^T is the transpose of *A*.

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $y := \alpha A^T x + \beta y$,
<i>m</i>	Number of block rows of the matrix <i>A</i> .
<i>k</i>	Number of block columns of the matrix <i>A</i> .
<i>lb</i>	Size of the block in the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by $lb * lb$.

Refer to *values* array description in [BSR Format](#) for more details.

indx

For one-based indexing, array containing the column indices plus one for each non-zero block of the matrix *A*. For zero-based indexing, array containing the column indices for each non-zero block of the matrix *A*.

Its length is equal to the number of non-zero blocks in the matrix *A*.

Refer to *columns* array description in [BSR Format](#) for more details.

pntrb

Array of length *m*.

This array contains row indices, such that *pntrb*[*i*] - *pntrb*[0] is the first index of block row *i* in the array *indx*

Refer to *pointerB* array description in [BSR Format](#) for more details.

pntre

Array of length *m*.

For zero-based indexing this array contains row indices, such that *pntre*[*i*] - *pntrb*[0] - 1 is the last index of block row *i* in the array *indx*.

Refer to *pointerE* array description in [BSR Format](#) for more details.

x

Array, size at least $(k \cdot lb)$ if *transa* = 'N' or 'n', and at least $(m \cdot lb)$ otherwise. On entry, the array *x* must contain the vector *x*.

beta

Specifies the scalar *beta*.

y

Array, size at least $(m \cdot lb)$ if *transa* = 'N' or 'n', and at least $(k \cdot lb)$ otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y

Overwritten by the updated vector *y*.

`mk1_cscmv`

Computes matrix-vector product for a sparse matrix in the CSC format (deprecated).

Syntax

```
void mk1_scscmv (const char *transa , const MKL_INT *m , const MKL_INT *k , const float
*alpha , const char *matdescra , const float *val , const MKL_INT *indx , const MKL_INT
*pntrb , const MKL_INT *pntre , const float *x , const float *beta , float *y );
```

```
void mk1_dcscmv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *indx , const
MKL_INT *pntrb , const MKL_INT *pntre , const double *x , const double *beta , double
*y );
```

```
void mk1_ccscmv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*indx , const MKL_INT *pntrb , const MKL_INT *pntre , const MKL_Complex8 *x , const
MKL_Complex8 *beta , MKL_Complex8 *y );
```

```
void mkl_zcscmv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*indx , const MKL_INT *pntrb , const MKL_INT *pntre , const MKL_Complex16 *x , const
MKL_Complex16 *beta , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?cscmv` routine performs a matrix-vector operation defined as

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

or

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

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed sparse column (CSC) format, *A*^T is the transpose of *A*.

NOTE

This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha A^T x + \beta y$,
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>k</i>	Number of columns of the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Array containing non-zero elements of the matrix <i>A</i> . Its length is <code>pntre[k-1] - pntrb[0]</code> . Refer to <i>values</i> array description in CSC Format for more details.

<i>indx</i>	<p>For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix <i>A</i>. For zero-based indexing, array containing the column indices for each non-zero element of the matrix <i>A</i>. Its length is equal to length of the <i>val</i> array.</p> <p>Refer to <i>rows</i> array description in CSC Format for more details.</p>
<i>pntrb</i>	<p>Array of length <i>k</i>.</p> <p>This array contains column indices, such that $pntrb[i] - pntrb[0] + 1$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerb</i> array description in CSC Format for more details.</p>
<i>pntre</i>	<p>Array of length <i>k</i>.</p> <p>For one-based indexing this array contains column indices, such that $pntre[i] - pntrb[1]$ is the last index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>For zero-based indexing this array contains column indices, such that $pntre[i] - pntrb[1] - 1$ is the last index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerE</i> array description in CSC Format for more details.</p>
<i>x</i>	<p>Array, size at least <i>k</i> if <i>transa</i> = 'N' or 'n' and at least <i>m</i> otherwise. On entry, the array <i>x</i> must contain the vector <i>x</i>.</p>
<i>beta</i>	<p>Specifies the scalar <i>beta</i>.</p>
<i>y</i>	<p>Array, size at least <i>m</i> if <i>transa</i> = 'N' or 'n' and at least <i>k</i> otherwise. On entry, the array <i>y</i> must contain the vector <i>y</i>.</p>

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

mkl_?coomv

Computes matrix - vector product for a sparse matrix in the coordinate format (deprecated).

Syntax

```
void mkl_scoomv (const char *transa , const MKL_INT *m , const MKL_INT *k , const float
*alpha , const char *matdescra , const float *val , const MKL_INT *rowind , const
MKL_INT *colind , const MKL_INT *nnz , const float *x , const float *beta , float *y );

void mkl_dcoomv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *rowind ,
const MKL_INT *colind , const MKL_INT *nnz , const double *x , const double *beta ,
double *y );

void mkl_ccoomv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex8 *x , const
MKL_Complex8 *beta , MKL_Complex8 *y );
```

```
void mkl_zcoomv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex16 *x , const
MKL_Complex16 *beta , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?coomv` routine performs a matrix-vector operation defined as

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

or

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

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed coordinate format, *A*^T is the transpose of *A*.

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha A^T x + \beta y$,
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>k</i>	Number of columns of the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	Array of length <i>nnz</i> .

For one-based indexing, contains the row indices plus one for each non-zero element of the matrix *A*.

For zero-based indexing, contains the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in [Coordinate Format](#) for more details.

colind

Array of length *nnz*.

For one-based indexing, contains the column indices plus one for each non-zero element of the matrix *A*.

For zero-based indexing, contains the column indices for each non-zero element of the matrix *A*.

Refer to *columns* array description in [Coordinate Format](#) for more details.

nnz

Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in [Coordinate Format](#) for more details.

x

Array, size at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.

beta

Specifies the scalar *beta*.

y

Array, size at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y

Overwritten by the updated vector *y*.

mkl_?csrsv

Solves a system of linear equations for a sparse matrix in the CSR format (deprecated).

Syntax

```
void mkl_scsrsv (const char *transa , const MKL_INT *m , const float *alpha , const
char *matdescra , const float *val , const MKL_INT *indx , const MKL_INT *pntrb , const
MKL_INT *pntre , const float *x , float *y );
```

```
void mkl_dcsrsv (const char *transa , const MKL_INT *m , const double *alpha , const
char *matdescra , const double *val , const MKL_INT *indx , const MKL_INT *pntrb ,
const MKL_INT *pntre , const double *x , double *y );
```

```
void mkl_ccsrsv (const char *transa , const MKL_INT *m , const MKL_Complex8 *alpha ,
const char *matdescra , const MKL_Complex8 *val , const MKL_INT *indx , const MKL_INT
*pntrb , const MKL_INT *pntre , const MKL_Complex8 *x , MKL_Complex8 *y );
```

```
void mkl_zcsrsv (const char *transa , const MKL_INT *m , const MKL_Complex16 *alpha ,
const char *matdescra , const MKL_Complex16 *val , const MKL_INT *indx , const MKL_INT
*pntrb , const MKL_INT *pntre , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSR format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(AT)*x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A*^T is the transpose of *A*.

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $y := \alpha \cdot \text{inv}(A) \cdot x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha \cdot \text{inv}(A^T) \cdot x$,
<i>m</i>	Number of columns of the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Array containing non-zero elements of the matrix <i>A</i> . Its length is <code>pntre[m - 1] - pntrb[0]</code> . Refer to <i>values</i> array description in CSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>indx</i>	For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix <i>A</i> . For zero-based indexing, array containing the column indices for each non-zero element of the matrix <i>A</i> .
-------------	---

Its length is equal to length of the *val* array.

Refer to *columns* array description in [CSR Format](#) for more details.

NOTE

Column indices must be sorted in increasing order for each row.

pntrb

Array of length *m*.

This array contains row indices, such that *pntrb*[*i*] - *pntrb*[0] is the first index of row *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in [CSR Format](#) for more details.

pntrE

Array of length *m*.

This array contains row indices, such that *pntrE*[*i*] - *pntrb*[0] - 1 is the last index of row *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in [CSR Format](#) for more details.

x

Array, size at least *m*.

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y

Array, size at least *m*.

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y

Contains solution vector *x*.

mkl_?bsrsv

Solves a system of linear equations for a sparse matrix in the BSR format (deprecated).

Syntax

```
void mkl_sbsrsv (const char *transa , const MKL_INT *m , const MKL_INT *lb , const
float *alpha , const char *matdescra , const float *val , const MKL_INT *indx , const
MKL_INT *pntrb , const MKL_INT *pntrE , const float *x , float *y );
```

```
void mkl_dbsrsv (const char *transa , const MKL_INT *m , const MKL_INT *lb , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *indx , const
MKL_INT *pntrb , const MKL_INT *pntrE , const double *x , double *y );
```

```
void mkl_cbsrsv (const char *transa , const MKL_INT *m , const MKL_INT *lb , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*indx , const MKL_INT *pntrb , const MKL_INT *pntrE , const MKL_Complex8 *x ,
MKL_Complex8 *y );
```

```
void mkl_zbsrsv (const char *transa , const MKL_INT *m , const MKL_INT *lb , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*indx , const MKL_INT *pntrb , const MKL_INT *pntrE , const MKL_Complex16 *x ,
MKL_Complex16 *y );
```

Include Files

- `mkl.h`

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?bsrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the BSR format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(AT)* x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A*^T is the transpose of *A*.

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then <i>y</i> := <i>alpha</i> *inv(<i>A</i>)* <i>x</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>y</i> := <i>alpha</i> *inv(<i>A</i> ^T)* <i>x</i> ,
<i>m</i>	Number of block columns of the matrix <i>A</i> .
<i>lb</i>	Size of the block in the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb</i> * <i>lb</i> . Refer to the <i>values</i> array description in BSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>indx</i>	<p>For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix <i>A</i>. For zero-based indexing, array containing the column indices for each non-zero element of the matrix <i>A</i>. Its length is equal to the number of non-zero blocks in the matrix <i>A</i>. Refer to the <i>columns</i> array description in BSR Format for more details.</p>
<i>pntrb</i>	<p>Array of length <i>m</i>.</p> <p>This array contains row indices, such that <i>pntrb</i>[<i>i</i>] - <i>pntrb</i>[0] is the first index of block row <i>i</i> in the array <i>indx</i>.</p> <p>Refer to <i>pointerB</i> array description in BSR Format for more details.</p>
<i>pntre</i>	<p>Array of length <i>m</i>.</p> <p>For one-based indexing this array contains row indices, such that <i>pntre</i>[<i>i</i>] - <i>pntrb</i>[1] is the last index of block row <i>i</i> in the array <i>indx</i>.</p> <p>For zero-based indexing this array contains row indices, such that <i>pntre</i>[<i>i</i>] - <i>pntrb</i>[0] - 1 is the last index of block row <i>i</i> in the array <i>indx</i>.</p> <p>Refer to <i>pointerE</i> array description in BSR Format for more details.</p>
<i>x</i>	<p>Array, size at least (<i>m*lb</i>).</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>. The elements are accessed with unit increment.</p>
<i>y</i>	<p>Array, size at least (<i>m*lb</i>).</p> <p>On entry, the array <i>y</i> must contain the vector <i>y</i>. The elements are accessed with unit increment.</p>

Output Parameters

<i>y</i>	Contains solution vector <i>x</i> .
----------	-------------------------------------

mkl_scscsv

Solves a system of linear equations for a sparse matrix in the CSC format (deprecated).

Syntax

```
void mkl_scscsv (const char *transa , const MKL_INT *m , const float *alpha , const
char *matdescra , const float *val , const MKL_INT *indx , const MKL_INT *pntrb , const
MKL_INT *pntre , const float *x , float *y );
```

```
void mkl_dcscsv (const char *transa , const MKL_INT *m , const double *alpha , const
char *matdescra , const double *val , const MKL_INT *indx , const MKL_INT *pntrb ,
const MKL_INT *pntrc , const double *x , double *y );

void mkl_ccscsv (const char *transa , const MKL_INT *m , const MKL_Complex8 *alpha ,
const char *matdescra , const MKL_Complex8 *val , const MKL_INT *indx , const MKL_INT
*pntrb , const MKL_INT *pntrc , const MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_zcscsv (const char *transa , const MKL_INT *m , const MKL_Complex16 *alpha ,
const char *matdescra , const MKL_Complex16 *val , const MKL_INT *indx , const MKL_INT
*pntrb , const MKL_INT *pntrc , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?cscsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSC format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(AT)* x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A^T* is the transpose of *A*.

NOTE

This routine supports a CSC format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha \cdot \text{inv}(A) \cdot x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha \cdot \text{inv}(A^T) \cdot x$,
<i>m</i>	Number of columns of the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Array containing non-zero elements of the matrix <i>A</i> . Its length is <code>pntrc[m-1] - pntrb[0]</code> .

Refer to *values* array description in [CSC Format](#) for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx

For one-based indexing, array containing the row indices plus one for each non-zero element of the matrix *A*.

For zero-based indexing, array containing the row indices for each non-zero element of the matrix *A*.

Its length is equal to length of the *val* array.

Refer to *columns* array description in [CSC Format](#) for more details.

NOTE

Row indices must be sorted in increasing order for each column.

pntrb

Array of length *m*.

This array contains column indices, such that $pntreb[i] - pntreb[0]$ is the first index of column *i* in the arrays *val* and *indx*.

Refer to *pointerb* array description in [CSC Format](#) for more details.

pntre

Array of length *m*.

This array contains column indices, such that $pntre[i] - pntreb[0] - 1$ is the last index of column *i* in the arrays *val* and *indx*.

Refer to *pointerE* array description in [CSC Format](#) for more details.

x

Array, size at least *m*.

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y

Array, size at least *m*.

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y

Contains the solution vector *x*.

mkl_?coosv

Solves a system of linear equations for a sparse matrix in the coordinate format (deprecated).

Syntax

```
void mkl_scoosv (const char *transa , const MKL_INT *m , const float *alpha , const
char *matdescra , const float *val , const MKL_INT *rowind , const MKL_INT *colind ,
const MKL_INT *nnz , const float *x , float *y );
```

```
void mkl_dcoosv (const char *transa , const MKL_INT *m , const double *alpha , const
char *matdescra , const double *val , const MKL_INT *rowind , const MKL_INT *colind ,
const MKL_INT *nnz , const double *x , double *y );
```

```
void mkl_ccoosv (const char *transa , const MKL_INT *m , const MKL_Complex8 *alpha ,
const char *matdescra , const MKL_Complex8 *val , const MKL_INT *rowind , const MKL_INT
*colind , const MKL_INT *nnz , const MKL_Complex8 *x , MKL_Complex8 *y );
```

```
void mkl_zcoosv (const char *transa , const MKL_INT *m , const MKL_Complex16 *alpha ,
const char *matdescra , const MKL_Complex16 *val , const MKL_INT *rowind , const
MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?coosv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the coordinate format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(AT)*x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A*^T is the transpose of *A*.

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $y := \alpha \cdot \text{inv}(A) \cdot x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha \cdot \text{inv}(A^T) \cdot x$,
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .

<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	<p>Array of length <i>nnz</i>, contains non-zero elements of the matrix <i>A</i> in the arbitrary order.</p> <p>Refer to <i>values</i> array description in Coordinate Format for more details.</p>
<i>rowind</i>	<p>Array of length <i>nnz</i>.</p> <p>For one-based indexing, contains the row indices plus one for each non-zero element of the matrix <i>A</i>.</p> <p>For zero-based indexing, contains the row indices for each non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>rows</i> array description in Coordinate Format for more details.</p>
<i>colind</i>	<p>Array of length <i>nnz</i>.</p> <p>For one-based indexing, contains the column indices plus one for each non-zero element of the matrix <i>A</i>.</p> <p>For zero-based indexing, contains the column indices for each non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>columns</i> array description in Coordinate Format for more details.</p>
<i>nnz</i>	<p>Specifies the number of non-zero element of the matrix <i>A</i>.</p> <p>Refer to <i>nnz</i> description in Coordinate Format for more details.</p>
<i>x</i>	<p>Array, size at least <i>m</i>.</p> <p>On entry, the array <i>x</i> must contain the vector <i>x</i>. The elements are accessed with unit increment.</p>
<i>y</i>	<p>Array, size at least <i>m</i>.</p> <p>On entry, the array <i>y</i> must contain the vector <i>y</i>. The elements are accessed with unit increment.</p>

Output Parameters

<i>y</i>	Contains solution vector <i>x</i> .
----------	-------------------------------------

mkl_?csrmm

Computes matrix - matrix product of a sparse matrix stored in the CSR format (deprecated).

Syntax

```
void mkl_scsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const float *alpha , const char *matdescra , const float *val , const
MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const float *b , const
MKL_INT *ldb , const float *beta , float *c , const MKL_INT *ldc );
```

```

void mkl_dcsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const double *alpha , const char *matdescra , const double *val , const
MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const double *b , const
MKL_INT *ldb , const double *beta , double *c , const MKL_INT *ldc );

void mkl_ccsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8
*val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const
MKL_Complex8 *b , const MKL_INT *ldb , const MKL_Complex8 *beta , MKL_Complex8 *c ,
const MKL_INT *ldc );

void mkl_zcsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16
*val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const
MKL_Complex16 *b , const MKL_INT *ldb , const MKL_Complex16 *beta , MKL_Complex16 *c ,
const MKL_INT *ldc );

```

Include Files

- mkl.h

Description

This routine is deprecated. Use [Use mkl_sparse_?_mm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrmm` routine performs a matrix-matrix operation defined as

$$C := \alpha A * B + \beta C$$

or

$$C := \alpha A^T * B + \beta C$$

or

$$C := \alpha A^H * B + \beta C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse row (CSR) format, A^T is the transpose of *A*, and A^H is the conjugate transpose of *A*.

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha A * B + \beta C$, If <i>transa</i> = 'T' or 't', then $C := \alpha A^T * B + \beta C$, If <i>transa</i> = 'C' or 'c', then $C := \alpha A^H * B + \beta C$.
<i>m</i>	Number of rows of the matrix <i>A</i> .

<i>n</i>	Number of columns of the matrix <i>C</i> .
<i>k</i>	Number of columns of the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	<p>Array containing non-zero elements of the matrix <i>A</i>.</p> <p>For zero-based indexing its length is <i>pntre</i>[<i>m</i>-1] - <i>pntrb</i>[0].</p> <p>Refer to <i>values</i> array description in CSR Format for more details.</p>
<i>indx</i>	<p>For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix <i>A</i>.</p> <p>For zero-based indexing, array containing the column indices for each non-zero element of the matrix <i>A</i>.</p> <p>Its length is equal to length of the <i>val</i> array.</p> <p>Refer to <i>columns</i> array description in CSR Format for more details.</p>
<i>pntrb</i>	<p>Array of length <i>m</i>.</p> <p>This array contains row indices, such that <i>pntrb</i>[<i>I</i>] - <i>pntrb</i>[0] is the first index of row <i>I</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerb</i> array description in CSR Format for more details.</p>
<i>pntre</i>	<p>Array of length <i>m</i>.</p> <p>This array contains row indices, such that <i>pntre</i>[<i>I</i>] - <i>pntrb</i>[0] - 1 is the last index of row <i>I</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerE</i> array description in CSR Format for more details.</p>
<i>b</i>	<p>Array, size <i>ldb</i> by at least <i>n</i> for non-transposed matrix <i>A</i> and at least <i>m</i> for transposed for one-based indexing, and (at least <i>k</i> for non-transposed matrix <i>A</i> and at least <i>m</i> for transposed, <i>ldb</i>) for zero-based indexing.</p> <p>On entry with <i>transa</i>='N' or 'n', the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.</p>
<i>ldb</i>	Specifies the leading dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.
<i>beta</i>	Specifies the scalar <i>beta</i> .
<i>c</i>	<p>Array, size <i>ldc</i> by <i>n</i> for one-based indexing, and (<i>m</i>, <i>ldc</i>) for zero-based indexing.</p> <p>On entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>.</p>

ldc Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c Overwritten by the matrix $(\alpha * A * B + \beta * C)$, $(\alpha * A^T * B + \beta * C)$, or $(\alpha * A^H * B + \beta * C)$.

mkl_?bsrmm

Computes matrix - matrix product of a sparse matrix stored in the BSR format (deprecated).

Syntax

```
void mkl_sbsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_INT *lb , const float *alpha , const char *matdescra , const
float *val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const
float *b , const MKL_INT *ldb , const float *beta , float *c , const MKL_INT *ldc );

void mkl_dbsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_INT *lb , const double *alpha , const char *matdescra , const
double *val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const
double *b , const MKL_INT *ldb , const double *beta , double *c , const MKL_INT *ldc );

void mkl_cbsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_INT *lb , const MKL_Complex8 *alpha , const char *matdescra ,
const MKL_Complex8 *val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT
*pntre , const MKL_Complex8 *b , const MKL_INT *ldb , const MKL_Complex8 *beta ,
MKL_Complex8 *c , const MKL_INT *ldc );

void mkl_zbsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_INT *lb , const MKL_Complex16 *alpha , const char *matdescra ,
const MKL_Complex16 *val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT
*pntre , const MKL_Complex16 *b , const MKL_INT *ldb , const MKL_Complex16 *beta ,
MKL_Complex16 *c , const MKL_INT *ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [Use mkl_sparse_?_mm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?bsrmm` routine performs a matrix-matrix operation defined as

$$C := \alpha * A * B + \beta * C$$

or

$$C := \alpha * A^T * B + \beta * C$$

or

$$C := \alpha * A^H * B + \beta * C,$$

where:

α and β are scalars,

B and C are dense matrices, A is an m -by- k sparse matrix in block sparse row (BSR) format, A^T is the transpose of A , and A^H is the conjugate transpose of A .

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	<p>Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-matrix product is computed as $C := \alpha * A * B + \beta * C$</p> <p>If <i>transa</i> = 'T' or 't', then the matrix-vector product is computed as $C := \alpha * A^T * B + \beta * C$</p> <p>If <i>transa</i> = 'C' or 'c', then the matrix-vector product is computed as $C := \alpha * A^H * B + \beta * C$,</p>
<i>m</i>	Number of block rows of the matrix A .
<i>n</i>	Number of columns of the matrix C .
<i>k</i>	Number of block columns of the matrix A .
<i>lb</i>	Size of the block in the matrix A .
<i>alpha</i>	Specifies the scalar α .
<i>matdescra</i>	<p>Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>Array containing elements of non-zero blocks of the matrix A. Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to the <i>values</i> array description in BSR Format for more details.</p>
<i>indx</i>	<p>For one-based indexing, array containing the column indices plus one for each non-zero block in the matrix A.</p> <p>For zero-based indexing, array containing the column indices for each non-zero block in the matrix A.</p> <p>Its length is equal to the number of non-zero blocks in the matrix A. Refer to the <i>columns</i> array description in BSR Format for more details.</p>
<i>pntrb</i>	<p>Array of length m.</p> <p>This array contains row indices, such that $pntrb[I] - pntrb[0]$ is the first index of block row I in the array <i>indx</i>.</p> <p>Refer to <i>pointerB</i> array description in BSR Format for more details.</p>
<i>pntrc</i>	Array of length m .

This array contains row indices, such that $pntre[I] - pntrb[0] - 1$ is the last index of block row I in the array $indx$.

Refer to *pointerE* array description in [BSR Format](#) for more details.

b

Array, size ldb by at least n for non-transposed matrix A and at least m for transposed for one-based indexing, and (at least k for non-transposed matrix A and at least m for transposed, ldb) for zero-based indexing.

On entry with $transa='N'$ or $'n'$, the leading n -by- k block part of the array b must contain the matrix B , otherwise the leading m -by- n block part of the array b must contain the matrix B .

ldb

Specifies the leading dimension (in blocks) of b as declared in the calling (sub)program.

beta

Specifies the scalar $beta$.

c

Array, size $ldc * n$ for one-based indexing, size $k * ldc$ for zero-based indexing.

On entry, the leading m -by- n block part of the array c must contain the matrix C , otherwise the leading n -by- k block part of the array c must contain the matrix C .

ldc

Specifies the leading dimension (in blocks) of c as declared in the calling (sub)program.

Output Parameters

c

Overwritten by the matrix $(alpha * A * B + beta * C)$ or $(alpha * A^T * B + beta * C)$ or $(alpha * A^H * B + beta * C)$.

mkl_?cscmm

Computes matrix-matrix product of a sparse matrix stored in the CSC format (deprecated).

Syntax

```
void mkl_scscmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const float *alpha , const char *matdescra , const float *val , const
MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const float *b , const
MKL_INT *ldb , const float *beta , float *c , const MKL_INT *ldc );
```

```
void mkl_dcscmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const double *alpha , const char *matdescra , const double *val , const
MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const double *b , const
MKL_INT *ldb , const double *beta , double *c , const MKL_INT *ldc );
```

```
void mkl_ccscmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8
*val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const
MKL_Complex8 *b , const MKL_INT *ldb , const MKL_Complex8 *beta , MKL_Complex8 *c ,
const MKL_INT *ldc );
```

```
void mkl_zcscmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16
*val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const
MKL_Complex16 *b , const MKL_INT *ldb , const MKL_Complex16 *beta , MKL_Complex16 *c ,
const MKL_INT *ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [Use mkl_sparse?_m](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?zcscmm` routine performs a matrix-matrix operation defined as

$$C := \alpha A * B + \beta C$$

or

$$C := \alpha A^T * B + \beta C,$$

or

$$C := \alpha A^H * B + \beta C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse column (CSC) format, A^T is the transpose of *A*, and A^H is the conjugate transpose of *A*.

NOTE

This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha A * B + \beta C$ If <i>transa</i> = 'T' or 't', then $C := \alpha A^T * B + \beta C$, If <i>transa</i> = 'C' or 'c', then $C := \alpha A^H * B + \beta C$
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>n</i>	Number of columns of the matrix <i>C</i> .
<i>k</i>	Number of columns of the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .

<i>val</i>	<p>Array containing non-zero elements of the matrix <i>A</i>.</p> <p>Its length is $pntrb[k-1] - pntrb[0]$.</p> <p>Refer to <i>values</i> array description in CSC Format for more details.</p>
<i>indx</i>	<p>For one-based indexing, array containing the row indices plus one for each non-zero element of the matrix <i>A</i>.</p> <p>For zero-based indexing, array containing the column indices for each non-zero element of the matrix <i>A</i>.</p> <p>Its length is equal to length of the <i>val</i> array.</p> <p>Refer to <i>rows</i> array description in CSC Format for more details.</p>
<i>pntrb</i>	<p>Array of length <i>k</i>.</p> <p>This array contains column indices, such that $pntrb[i] - pntrb[0]$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerb</i> array description in CSC Format for more details.</p>
<i>pntrc</i>	<p>Array of length <i>k</i>.</p> <p>This array contains column indices, such that $pntrc[i] - pntrb[0] - 1$ is the last index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerE</i> array description in CSC Format for more details.</p>
<i>b</i>	<p>Array, size <i>ldb</i> by at least <i>n</i> for non-transposed matrix <i>A</i> and at least <i>m</i> for transposed for one-based indexing, and (at least <i>k</i> for non-transposed matrix <i>A</i> and at least <i>m</i> for transposed, <i>ldb</i>) for zero-based indexing.</p> <p>On entry with <i>transa</i> = 'N' or 'n', the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>, otherwise the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.</p>
<i>ldb</i>	<p>Specifies the leading dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.</p>
<i>beta</i>	<p>Specifies the scalar <i>beta</i>.</p>
<i>c</i>	<p>Array, size <i>ldc</i> by <i>n</i> for one-based indexing, and (<i>m</i>, <i>ldc</i>) for zero-based indexing.</p> <p>On entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>, otherwise the leading <i>k</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>.</p>
<i>ldc</i>	<p>Specifies the leading dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.</p>

Output Parameters

<i>c</i>	Overwritten by the matrix $(\alpha * A * B + \beta * C)$ or $(\alpha * A^T * B + \beta * C)$ or $(\alpha * A^H * B + \beta * C)$.
----------	--

mkl_?coomm

Computes matrix-matrix product of a sparse matrix stored in the coordinate format (deprecated).

Syntax

```
void mkl_scoomm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const float *alpha , const char *matdescra , const float *val , const
MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const float *b , const
MKL_INT *ldb , const float *beta , float *c , const MKL_INT *ldc );

void mkl_dcoomm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const double *alpha , const char *matdescra , const double *val , const
MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const double *b , const
MKL_INT *ldb , const double *beta , double *c , const MKL_INT *ldc );

void mkl_ccoomm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8
*val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const
MKL_Complex8 *b , const MKL_INT *ldb , const MKL_Complex8 *beta , MKL_Complex8 *c ,
const MKL_INT *ldc );

void mkl_zcoomm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16
*val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const
MKL_Complex16 *b , const MKL_INT *ldb , const MKL_Complex16 *beta , MKL_Complex16 *c ,
const MKL_INT *ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [Use mkl_sparse_?_mm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?coomm` routine performs a matrix-matrix operation defined as

$$C := \alpha * A * B + \beta * C$$

or

$$C := \alpha * A^T * B + \beta * C,$$

or

$$C := \alpha * A^H * B + \beta * C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the coordinate format, A^T is the transpose of *A*, and A^H is the conjugate transpose of *A*.

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	<p>Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$</p> <p>If <i>transa</i> = 'T' or 't', then $C := \alpha * A^T * B + \beta * C$,</p> <p>If <i>transa</i> = 'C' or 'c', then $C := \alpha * A^H * B + \beta * C$.</p>
<i>m</i>	Number of rows of the matrix A.
<i>n</i>	Number of columns of the matrix C.
<i>k</i>	Number of columns of the matrix A.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	<p>Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>Array of length <i>nnz</i>, contains non-zero elements of the matrix A in the arbitrary order.</p> <p>Refer to <i>values</i> array description in Coordinate Format for more details.</p>
<i>rowind</i>	<p>Array of length <i>nnz</i>.</p> <p>For one-based indexing, contains the row indices plus one for each non-zero element of the matrix A.</p> <p>For zero-based indexing, contains the row indices for each non-zero element of the matrix A.</p> <p>Refer to <i>rows</i> array description in Coordinate Format for more details.</p>
<i>colind</i>	<p>Array of length <i>nnz</i>.</p> <p>For one-based indexing, contains the column indices plus one for each non-zero element of the matrix A.</p> <p>For zero-based indexing, contains the column indices for each non-zero element of the matrix A.</p> <p>Refer to <i>columns</i> array description in Coordinate Format for more details.</p>
<i>nnz</i>	<p>Specifies the number of non-zero element of the matrix A.</p> <p>Refer to <i>nnz</i> description in Coordinate Format for more details.</p>
<i>b</i>	<p>Array, size <i>ldb</i> by at least <i>n</i> for non-transposed matrix A and at least <i>m</i> for transposed for one-based indexing, and (at least <i>k</i> for non-transposed matrix A and at least <i>m</i> for transposed, <i>ldb</i>) for zero-based indexing.</p> <p>On entry with <i>transa</i> = 'N' or 'n', the leading <i>k</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix B, otherwise the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix B.</p>

<i>ldb</i>	Specifies the leading dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.
<i>beta</i>	Specifies the scalar <i>beta</i> .
<i>c</i>	Array, size <i>ldc</i> by <i>n</i> for one-based indexing, and (<i>m</i> , <i>ldc</i>) for zero-based indexing. On entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> .
<i>ldc</i>	Specifies the leading dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Overwritten by the matrix $(\alpha * A * B + \beta * C)$, $(\alpha * A^T * B + \beta * C)$, or $(\alpha * A^H * B + \beta * C)$.
----------	---

mkl_?csrsm

Solves a system of linear matrix equations for a sparse matrix in the CSR format (deprecated).

Syntax

```
void mkl_scsrsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const float
*alpha , const char *matdescra , const float *val , const MKL_INT *indx , const MKL_INT
*pntrb , const MKL_INT *pntre , const float *b , const MKL_INT *ldb , float *c , const
MKL_INT *ldc );
```

```
void mkl_dcsrsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *indx , const
MKL_INT *pntrb , const MKL_INT *pntre , const double *b , const MKL_INT *ldb , double
*c , const MKL_INT *ldc );
```

```
void mkl_ccsrsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*indx , const MKL_INT *pntrb , const MKL_INT *pntre , const MKL_Complex8 *b , const
MKL_INT *ldb , MKL_Complex8 *c , const MKL_INT *ldc );
```

```
void mkl_zcsrsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*indx , const MKL_INT *pntrb , const MKL_INT *pntre , const MKL_Complex16 *b , const
MKL_INT *ldb , MKL_Complex16 *c , const MKL_INT *ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_trsm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSR format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(AT)*B,
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A*^T is the transpose of *A*.

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $C := \alpha \cdot \text{inv}(A) \cdot B$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha \cdot \text{inv}(A^T) \cdot B$,
<i>m</i>	Number of columns of the matrix <i>A</i> .
<i>n</i>	Number of columns of the matrix <i>C</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Array containing non-zero elements of the matrix <i>A</i> . For zero-based indexing its length is <code>pntre[m-1] - pntrb[0]</code> . Refer to <i>values</i> array description in CSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>indx</i>	For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix <i>A</i> . For zero-based indexing, array containing the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to length of the <i>val</i> array. Refer to <i>columns</i> array description in CSR Format for more details.
-------------	---

NOTE

Column indices must be sorted in increasing order for each row.

<i>pntrb</i>	<p>Array of length m.</p> <p>This array contains row indices, such that $pntrb[i] - pntrb[0]$ is the first index of row i in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerb</i> array description in CSR Format for more details.</p>
<i>pntrc</i>	<p>Array of length m.</p> <p>For zero-based indexing this array contains row indices, such that $pntrc[i] - pntrb[0] - 1$ is the last index of row i in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerE</i> array description in CSR Format for more details.</p>
<i>b</i>	<p>Array, size $ldb * n$ for one-based indexing, and (m, ldb) for zero-based indexing.</p> <p>On entry the leading m-by-n part of the array <i>b</i> must contain the matrix <i>B</i>.</p>
<i>ldb</i>	<p>Specifies the leading dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.</p>
<i>ldc</i>	<p>Specifies the leading dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.</p>

Output Parameters

<i>c</i>	<p>Array, size ldc by n for one-based indexing, and (m, ldc) for zero-based indexing.</p> <p>The leading m-by-n part of the array <i>c</i> contains the output matrix <i>C</i>.</p>
----------	--

mkl_scscsm

Solves a system of linear matrix equations for a sparse matrix in the CSC format (deprecated).

Syntax

```
void mkl_scscsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const float
*alpha , const char *matdescra , const float *val , const MKL_INT *indx , const MKL_INT
*pntrb , const MKL_INT *pntrc , const float *b , const MKL_INT *ldb , float *c , const
MKL_INT *ldc );
```

```
void mkl_dcscsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *indx , const
MKL_INT *pntrb , const MKL_INT *pntrc , const double *b , const MKL_INT *ldb , double
*c , const MKL_INT *ldc );
```

```
void mkl_ccscsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*indx , const MKL_INT *pntrb , const MKL_INT *pntre , const MKL_Complex8 *b , const
MKL_INT *ldb , MKL_Complex8 *c , const MKL_INT *ldc );

void mkl_zcscsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*indx , const MKL_INT *pntrb , const MKL_INT *pntre , const MKL_Complex16 *b , const
MKL_INT *ldb , MKL_Complex16 *c , const MKL_INT *ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse?_trsm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?cscsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSC format:

$$C := \alpha * \text{inv}(A) * B$$

or

$$C := \alpha * \text{inv}(A^T) * B,$$

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of *A*.

NOTE

This routine supports a CSC format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	Specifies the system of equations. If <i>transa</i> = 'N' or 'n', then $C := \alpha * \text{inv}(A) * B$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha * \text{inv}(A^T) * B$,
<i>m</i>	Number of columns of the matrix <i>A</i> .
<i>n</i>	Number of columns of the matrix <i>C</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Array containing non-zero elements of the matrix <i>A</i> . For zero-based indexing its length is $pntre[m] - pntrb[0]$.

Refer to *values* array description in [CSC Format](#) for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx

For one-based indexing, array containing the row indices plus one for each non-zero element of the matrix *A*. For zero-based indexing, array containing the row indices for each non-zero element of the matrix *A*.

Refer to *rows* array description in [CSC Format](#) for more details.

NOTE

Row indices must be sorted in increasing order for each column.

pntrb

Array of length *m*.

This array contains column indices, such that *pntrb*[*I*] - *pntrb*[0] is the first index of column *I* in the arrays *val* and *indx*.

Refer to *pointerb* array description in [CSC Format](#) for more details.

pntrb

Array of length *m*.

This array contains column indices, such that *pntrb*[*I*] - *pntrb*[1]-1 is the last index of column *I* in the arrays *val* and *indx*.

Refer to *pointerE* array description in [CSC Format](#) for more details.

b

Array, size *ldb* by *n* for one-based indexing, and (*m*, *ldb*) for zero-based indexing.

On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

ldc

Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c

Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.

The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

mkl_?coosm

Solves a system of linear matrix equations for a sparse matrix in the coordinate format (deprecated).

Syntax

```
void mkl_scoosm (const char *transa , const MKL_INT *m , const MKL_INT *n , const float
*alpha , const char *matdescra , const float *val , const MKL_INT *rowind , const
MKL_INT *colind , const MKL_INT *nnz , const float *b , const MKL_INT *ldb , float *c ,
const MKL_INT *ldc );
```

```
void mkl_dcoosm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *rowind ,
const MKL_INT *colind , const MKL_INT *nnz , const double *b , const MKL_INT *ldb ,
double *c , const MKL_INT *ldc );
```

```
void mkl_ccoosm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex8 *b , const
MKL_INT *ldb , MKL_Complex8 *c , const MKL_INT *ldc );
```

```
void mkl_zcoosm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex16 *b , const
MKL_INT *ldb , MKL_Complex16 *c , const MKL_INT *ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_trsm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?coosm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the coordinate format:

$$C := \alpha \cdot \text{inv}(A) \cdot B$$

or

$$C := \alpha \cdot \text{inv}(A^T) \cdot B,$$

where:

α is scalar, B and C are dense matrices, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A .

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

transa

Specifies the system of linear equations.

If *transa* = 'N' or 'n', then the matrix-matrix product is computed as

$$C := \alpha \cdot \text{inv}(A) \cdot B$$

If $transa = 'T'$ or $'t'$ or $'C'$ or $'c'$, then the matrix-vector product is computed as $C := \alpha * \text{inv}(A^T) * B$,

<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>n</i>	Number of columns of the matrix <i>C</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	Array of length <i>nnz</i> . For one-based indexing, contains the row indices plus one for each non-zero element of the matrix <i>A</i> . For zero-based indexing, contains the row indices for each non-zero element of the matrix <i>A</i> . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	Array of length <i>nnz</i> . For one-based indexing, contains the column indices plus one for each non-zero element of the matrix <i>A</i> . For zero-based indexing, contains the row indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	Specifies the number of non-zero element of the matrix <i>A</i> . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>b</i>	Array, size <i>ldb</i> by <i>n</i> for one-based indexing, and (<i>m</i> , <i>ldb</i>) for zero-based indexing. Before entry the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>ldb</i>	Specifies the leading dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.
<i>ldc</i>	Specifies the leading dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Array, size <i>ldc</i> by <i>n</i> for one-based indexing, and (<i>m</i> , <i>ldc</i>) for zero-based indexing.
----------	---

The leading m -by- n part of the array c contains the output matrix C .

mkl_?bsrsm

Solves a system of linear matrix equations for a sparse matrix in the BSR format (deprecated).

Syntax

```
void mkl_sbsrsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *lb , const float *alpha , const char *matdescra , const float *val , const
MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const float *b , const
MKL_INT *ldb , float *c , const MKL_INT *ldc );

void mkl_dbsrsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *lb , const double *alpha , const char *matdescra , const double *val , const
MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const double *b , const
MKL_INT *ldb , double *c , const MKL_INT *ldc );

void mkl_cbsrsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *lb , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8
*val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const
MKL_Complex8 *b , const MKL_INT *ldb , MKL_Complex8 *c , const MKL_INT *ldc );

void mkl_zbsrsm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *lb , const MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16
*val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const
MKL_Complex16 *b , const MKL_INT *ldb , MKL_Complex16 *c , const MKL_INT *ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_trsm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?bsrsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the BSR format:

$$C := \alpha * \text{inv}(A) * B$$

or

$$C := \alpha * \text{inv}(A^T) * B,$$

where:

α is scalar, B and C are dense matrices, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A .

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

<i>transa</i>	<p>Specifies the operation.</p> <p>If <i>transa</i> = 'N' or 'n', then the matrix-matrix product is computed as $C := \alpha * \text{inv}(A) * B$.</p> <p>If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $C := \alpha * \text{inv}(A^T) * B$.</p>
<i>m</i>	Number of block columns of the matrix A.
<i>n</i>	Number of columns of the matrix C.
<i>lb</i>	Size of the block in the matrix A.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	<p>Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>".</p>
<i>val</i>	<p>Array containing elements of non-zero blocks of the matrix A. Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to the <i>values</i> array description in BSR Format for more details.</p>

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>indx</i>	<p>For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix A. For zero-based indexing, array containing the column indices for each non-zero element of the matrix A.</p> <p>Its length is equal to the number of non-zero blocks in the matrix A.</p> <p>Refer to the <i>columns</i> array description in BSR Format for more details.</p>
<i>pntrb</i>	<p>Array of length <i>m</i>.</p> <p>This array contains row indices, such that $pntrb[i] - pntrb[0]$ is the first index of block row <i>i</i> in the array <i>indx</i>.</p> <p>Refer to <i>pointerB</i> array description in BSR Format for more details.</p>
<i>pntre</i>	<p>Array of length <i>m</i>.</p> <p>This array contains row indices, such that $pntre[i] - pntrb[0] - 1$ is the last index of block row <i>i</i> in the arrays <i>val</i> and <i>indx</i>.</p> <p>Refer to <i>pointerE</i> array description in BSR Format for more details.</p>
<i>b</i>	<p>Array, size $ldb * n$ for one-based indexing, size $m * ldb$ for zero-based indexing.</p> <p>On entry the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix B.</p>

<i>ldb</i>	Specifies the leading dimension (in blocks) of <i>b</i> as declared in the calling (sub)program.
<i>ldc</i>	Specifies the leading dimension (in blocks) of <i>c</i> as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Array, size <i>ldc</i> * <i>n</i> for one-based indexing, size <i>m</i> * <i>ldc</i> for zero-based indexing. The leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> contains the output matrix <i>C</i> .
----------	--

mkl_?diamv

Computes matrix - vector product for a sparse matrix in the diagonal format with one-based indexing (deprecated).

Syntax

```
void mkl_sdiamv (const char *transa , const MKL_INT *m , const MKL_INT *k , const float
*alpha , const char *matdescra , const float *val , const MKL_INT *lval , const MKL_INT
*idiag , const MKL_INT *ndiag , const float *x , const float *beta , float *y );

void mkl_ddiamv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *lval , const
MKL_INT *idiag , const MKL_INT *ndiag , const double *x , const double *beta , double
*y );

void mkl_cdiamv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex8 *x , const
MKL_Complex8 *beta , MKL_Complex8 *y );

void mkl_zdiamv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex16 *x , const
MKL_Complex16 *beta , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?diamv` routine performs a matrix-vector operation defined as

$$y := \alpha A x + \beta y$$

or

$$y := \alpha A^T x + \beta y,$$

where:

alpha and *beta* are scalars,

x and y are vectors,

A is an m -by- k sparse matrix stored in the diagonal format, A^T is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha * A * x + \beta * y$, If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha * A^T * x + \beta * y$.
<i>m</i>	Number of rows of the matrix A .
<i>k</i>	Number of columns of the matrix A .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix A . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix A . Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	Specifies the number of non-zero diagonals of the matrix A .
<i>x</i>	Array, size at least k if <i>transa</i> = 'N' or 'n', and at least m otherwise. On entry, the array <i>x</i> must contain the vector x .
<i>beta</i>	Specifies the scalar <i>beta</i> .
<i>y</i>	Array, size at least m if <i>transa</i> = 'N' or 'n', and at least k otherwise. On entry, the array <i>y</i> must contain the vector y .

Output Parameters

<i>y</i>	Overwritten by the updated vector y .
----------	---

mkl_?skymv

Computes matrix - vector product for a sparse matrix in the skyline storage format with one-based indexing (deprecated).

Syntax

```
void mkl_sskymv (const char *transa , const MKL_INT *m , const MKL_INT *k , const float
*alpha , const char *matdescra , const float *val , const MKL_INT *pntr , const float
*x , const float *beta , float *y );
```

```
void mkl_dskymv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *pntr , const
double *x , const double *beta , double *y );
```

```
void mkl_cskymv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*pntr , const MKL_Complex8 *x , const MKL_Complex8 *beta , MKL_Complex8 *y );
```

```
void mkl_zskymv (const char *transa , const MKL_INT *m , const MKL_INT *k , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*pntr , const MKL_Complex16 *x , const MKL_Complex16 *beta , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_mv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?skymv` routine performs a matrix-vector operation defined as

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

or

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

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix stored using the skyline storage scheme, *A*^T is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

transa

Specifies the operation.

If *transa* = 'N' or 'n', then $y := \alpha A x + \beta y$

If *transa* = 'T' or 't' or 'C' or 'c', then $y := \alpha A^T x + \beta y$,

m

Number of rows of the matrix *A*.

<i>k</i>	Number of columns of the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .

NOTE

General matrices (*matdescra*[0]='G') is not supported.

<i>val</i>	<p>Array containing the set of elements of the matrix <i>A</i> in the skyline profile form.</p> <p>If <i>matdescrsa</i>[1]= 'L', then <i>val</i> contains elements from the low triangle of the matrix <i>A</i>.</p> <p>If <i>matdescrsa</i>[1]= 'U', then <i>val</i> contains elements from the upper triangle of the matrix <i>A</i>.</p> <p>Refer to <i>values</i> array description in Skyline Storage Scheme for more details.</p>
<i>pntr</i>	<p>Array of length (<i>m</i> + 1) for lower triangle, and (<i>k</i> + 1) for upper triangle.</p> <p>It contains the indices specifying in the <i>val</i> the positions of the first element in each row (column) of the matrix <i>A</i>. Refer to <i>pointers</i> array description in Skyline Storage Scheme for more details.</p>
<i>x</i>	Array, size at least <i>k</i> if <i>transa</i> = 'N' or 'n' and at least <i>m</i> otherwise. On entry, the array <i>x</i> must contain the vector <i>x</i> .
<i>beta</i>	Specifies the scalar <i>beta</i> .
<i>y</i>	Array, size at least <i>m</i> if <i>transa</i> = 'N' or 'n' and at least <i>k</i> otherwise. On entry, the array <i>y</i> must contain the vector <i>y</i> .

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

[mkl_?diasv](#)

Solves a system of linear equations for a sparse matrix in the diagonal format with one-based indexing (deprecated).

Syntax

```
void mkl_sdiasv (const char *transa , const MKL_INT *m , const float *alpha , const
char *matdescra , const float *val , const MKL_INT *lval , const MKL_INT *idiag , const
MKL_INT *ndiag , const float *x , float *y );

void mkl_ddiasv (const char *transa , const MKL_INT *m , const double *alpha , const
char *matdescra , const double *val , const MKL_INT *lval , const MKL_INT *idiag ,
const MKL_INT *ndiag , const double *x , double *y );
```

```
void mkl_cdiasv (const char *transa , const MKL_INT *m , const MKL_Complex8 *alpha ,
const char *matdescra , const MKL_Complex8 *val , const MKL_INT *lval , const MKL_INT
*idiag , const MKL_INT *ndiag , const MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_zdiasv (const char *transa , const MKL_INT *m , const MKL_Complex16 *alpha ,
const char *matdescra , const MKL_Complex16 *val , const MKL_INT *lval , const MKL_INT
*idiag , const MKL_INT *ndiag , const MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?diasv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(AT)* x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A*^T is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $y := \alpha \cdot \text{inv}(A) \cdot x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha \cdot \text{inv}(A^T) \cdot x$,
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.

idiag Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*.

NOTE

All elements of this array must be sorted in increasing order.

Refer to *distance* array description in [Diagonal Storage Scheme](#) for more details.

ndiag Specifies the number of non-zero diagonals of the matrix *A*.

x Array, size at least *m*.

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y Array, size at least *m*.

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y Contains solution vector *x*.

[mkl_?skysv](#)

Solves a system of linear equations for a sparse matrix in the skyline format with one-based indexing (deprecated).

Syntax

```
void mkl_sskysv (const char *transa , const MKL_INT *m , const float *alpha , const
char *matdescra , const float *val , const MKL_INT *pntr , const float *x , float *y );

void mkl_dskysv (const char *transa , const MKL_INT *m , const double *alpha , const
char *matdescra , const double *val , const MKL_INT *pntr , const double *x , double
*y );

void mkl_cskysv (const char *transa , const MKL_INT *m , const MKL_Complex8 *alpha ,
const char *matdescra , const MKL_Complex8 *val , const MKL_INT *pntr , const
MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_zskysv (const char *transa , const MKL_INT *m , const MKL_Complex16 *alpha ,
const char *matdescra , const MKL_Complex16 *val , const MKL_INT *pntr , const
MKL_Complex16 *x , MKL_Complex16 *y );
```

Include Files

- `mkl.h`

Description

This routine is deprecated. Use [mkl_sparse_?_trsv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?skysv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the skyline storage format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(AT)*x,
```

where:

α is scalar, x and y are vectors, A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A^T is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $y := \alpha \cdot \text{inv}(A) \cdot x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha \cdot \text{inv}(A^T) \cdot x$,
<i>m</i>	Number of rows of the matrix A .
<i>alpha</i>	Specifies the scalar α .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .

NOTE

General matrices (*matdescra*[0]='G') is not supported.

<i>val</i>	Array containing the set of elements of the matrix A in the skyline profile form. If <i>matdescra</i> [2]= 'L', then <i>val</i> contains elements from the low triangle of the matrix A . If <i>matdescra</i> [2]= 'U', then <i>val</i> contains elements from the upper triangle of the matrix A . Refer to <i>values</i> array description in Skyline Storage Scheme for more details.
<i>pntr</i>	Array of length $(m + 1)$ for lower triangle, and $(k + 1)$ for upper triangle. It contains the indices specifying in the <i>val</i> the positions of the first element in each row (column) of the matrix A . Refer to <i>pointers</i> array description in Skyline Storage Scheme for more details.
<i>x</i>	Array, size at least m .

On entry, the array x must contain the vector x . The elements are accessed with unit increment.

y

Array, size at least m .

On entry, the array y must contain the vector y . The elements are accessed with unit increment.

Output Parameters

y

Contains solution vector x .

mkl_?diamm

Computes matrix-matrix product of a sparse matrix stored in the diagonal format with one-based indexing (deprecated).

Syntax

```
void mkl_sdiamm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const float *alpha , const char *matdescra , const float *val , const
MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const float *b , const
MKL_INT *ldb , const float *beta , float *c , const MKL_INT *ldc );
```

```
void mkl_ddiamm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const double *alpha , const char *matdescra , const double *val , const
MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const double *b , const
MKL_INT *ldb , const double *beta , double *c , const MKL_INT *ldc );
```

```
void mkl_cdiamm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8
*val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const
MKL_Complex8 *b , const MKL_INT *ldb , const MKL_Complex8 *beta , MKL_Complex8 *c ,
const MKL_INT *ldc );
```

```
void mkl_zdiamm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16
*val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const
MKL_Complex16 *b , const MKL_INT *ldb , const MKL_Complex16 *beta , MKL_Complex16 *c ,
const MKL_INT *ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [Use mkl_sparse_?_mm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?diamm` routine performs a matrix-matrix operation defined as

```
 $C := \alpha * A * B + \beta * C$ 
```

or

```
 $C := \alpha * A^T * B + \beta * C,$ 
```

or

```
 $C := \alpha * A^H * B + \beta * C,$ 
```

where:

α and β are scalars,

B and C are dense matrices, A is an m -by- k sparse matrix in the diagonal format, A^T is the transpose of A , and A^H is the conjugate transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$, If <i>transa</i> = 'T' or 't', then $C := \alpha * A^T * B + \beta * C$, If <i>transa</i> = 'C' or 'c', then $C := \alpha * A^H * B + \beta * C$.
<i>m</i>	Number of rows of the matrix A .
<i>n</i>	Number of columns of the matrix C .
<i>k</i>	Number of columns of the matrix A .
<i>alpha</i>	Specifies the scalar α .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix A . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix A . Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	Specifies the number of non-zero diagonals of the matrix A .
<i>b</i>	Array, size $ldb * n$. On entry with <i>transa</i> = 'N' or 'n', the leading k -by- n part of the array <i>b</i> must contain the matrix B , otherwise the leading m -by- n part of the array <i>b</i> must contain the matrix B .

<i>ldb</i>	Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program.
<i>beta</i>	Specifies the scalar <i>beta</i> .
<i>c</i>	Array, size <i>ldc</i> by <i>n</i> . On entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> .
<i>ldc</i>	Specifies the leading dimension of <i>c</i> as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Overwritten by the matrix $(\alpha * A * B + \beta * C)$, $(\alpha * A^T * B + \beta * C)$, or $(\alpha * A^H * B + \beta * C)$.
----------	---

mkl_?skymm

Computes matrix-matrix product of a sparse matrix stored using the skyline storage scheme with one-based indexing (deprecated).

Syntax

```
void mkl_sskymm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const float *alpha , const char *matdescra , const float *val , const
MKL_INT *pntr , const float *b , const MKL_INT *ldb , const float *beta , float *c ,
const MKL_INT *ldc );

void mkl_dskymm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const double *alpha , const char *matdescra , const double *val , const
MKL_INT *pntr , const double *b , const MKL_INT *ldb , const double *beta , double *c ,
const MKL_INT *ldc );

void mkl_cskymm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8
*val , const MKL_INT *pntr , const MKL_Complex8 *b , const MKL_INT *ldb , const
MKL_Complex8 *beta , MKL_Complex8 *c , const MKL_INT *ldc );

void mkl_zskymm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , const MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16
*val , const MKL_INT *pntr , const MKL_Complex16 *b , const MKL_INT *ldb , const
MKL_Complex16 *beta , MKL_Complex16 *c , const MKL_INT *ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [Use mkl_sparse_?_mm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?skymm` routine performs a matrix-matrix operation defined as

$$C := \alpha * A * B + \beta * C$$

or

```
 $C := \alpha * A^T * B + \beta * C,$ 
```

or

```
 $C := \alpha * A^H * B + \beta * C,$ 
```

where:

α and β are scalars,

B and C are dense matrices, A is an m -by- k sparse matrix in the skyline storage format, A^T is the transpose of A , and A^H is the conjugate transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>transa</i>	Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha * A * B + \beta * C$, If <i>transa</i> = 'T' or 't', then $C := \alpha * A^T * B + \beta * C$, If <i>transa</i> = 'C' or 'c', then $C := \alpha * A^H * B + \beta * C$.
<i>m</i>	Number of rows of the matrix A .
<i>n</i>	Number of columns of the matrix C .
<i>k</i>	Number of columns of the matrix A .
<i>alpha</i>	Specifies the scalar α .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .

NOTE

General matrices (*matdescra* [0]='G') is not supported.

<i>val</i>	Array containing the set of elements of the matrix A in the skyline profile form. If <i>matdescra</i> [2] = 'L', then <i>val</i> contains elements from the low triangle of the matrix A . If <i>matdescra</i> [2] = 'U', then <i>val</i> contains elements from the upper triangle of the matrix A . Refer to <i>values</i> array description in Skyline Storage Scheme for more details.
<i>pntr</i>	Array of length $(m + 1)$ for lower triangle, and $(k + 1)$ for upper triangle.

It contains the indices specifying the positions of the first element of the matrix A in each row (for the lower triangle) or column (for upper triangle) in the `val` array such that `val[pntr[i] - 1]` is the first element in row or column $i + 1$. Refer to `pointers` array description in [Skyline Storage Scheme](#) for more details.

<code>b</code>	<p>Array, size <code>ldb * n</code>.</p> <p>On entry with <code>transa = 'N'</code> or <code>'n'</code>, the leading k-by-n part of the array <code>b</code> must contain the matrix B, otherwise the leading m-by-n part of the array <code>b</code> must contain the matrix B.</p>
<code>ldb</code>	Specifies the leading dimension of <code>b</code> as declared in the calling (sub)program.
<code>beta</code>	Specifies the scalar <code>beta</code> .
<code>c</code>	<p>Array, size <code>ldc</code> by <code>n</code>.</p> <p>On entry, the leading m-by-n part of the array <code>c</code> must contain the matrix C, otherwise the leading k-by-n part of the array <code>c</code> must contain the matrix C.</p>
<code>ldc</code>	Specifies the leading dimension of <code>c</code> as declared in the calling (sub)program.

Output Parameters

<code>c</code>	Overwritten by the matrix $(\alpha * A * B + \beta * C)$, $(\alpha * A^T * B + \beta * C)$, or $(\alpha * A^H * B + \beta * C)$.
----------------	---

`mkl_sdiasm`

Solves a system of linear matrix equations for a sparse matrix in the diagonal format with one-based indexing (deprecated).

Syntax

```
void mkl_sdiasm (const char *transa , const MKL_INT *m , const MKL_INT *n , const float
*alpha , const char *matdescra , const float *val , const MKL_INT *lval , const MKL_INT
*idiag , const MKL_INT *ndiag , const float *b , const MKL_INT *ldb , float *c , const
MKL_INT *ldc );
```

```
void mkl_ddiasm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *lval , const
MKL_INT *idiag , const MKL_INT *ndiag , const double *b , const MKL_INT *ldb , double
*c , const MKL_INT *ldc );
```

```
void mkl_cdiasm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex8 *b , const
MKL_INT *ldb , MKL_Complex8 *c , const MKL_INT *ldc );
```

```
void mkl_zdiasm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex16 *b , const
MKL_INT *ldb , MKL_Complex16 *c , const MKL_INT *ldc );
```

Include Files

- `mkl.h`

Description

This routine is deprecated. Use `mkl_sparse_?_trsm` from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?diasm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the diagonal format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(AT)*B,
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A*^T is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then <i>C</i> := <i>alpha</i> *inv(<i>A</i>)* <i>B</i> , If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>C</i> := <i>alpha</i> *inv(<i>A</i> ^T)* <i>B</i> .
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>n</i>	Number of columns of the matrix <i>C</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	Leading dimension of <i>val</i> , <i>lval</i> ≥ <i>m</i> . Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> .

NOTE

All elements of this array must be sorted in increasing order.

Refer to *distance* array description in [Diagonal Storage Scheme](#) for more details.

ndiag

Specifies the number of non-zero diagonals of the matrix *A*.

b

Array, size *ldb** *n*.

On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

Specifies the leading dimension of *b* as declared in the calling (sub)program.

ldc

Specifies the leading dimension of *c* as declared in the calling (sub)program.

Output Parameters

c

Array, size *ldc* by *n*.

The leading *m*-by-*n* part of the array *c* contains the matrix *C*.

mkl_?skysm

Solves a system of linear matrix equations for a sparse matrix stored using the skyline storage scheme with one-based indexing (deprecated).

Syntax

```
void mkl_sskysm (const char *transa , const MKL_INT *m , const MKL_INT *n , const float
*alpha , const char *matdescra , const float *val , const MKL_INT *pntr , const float
*b , const MKL_INT *ldb , float *c , const MKL_INT *ldc );
```

```
void mkl_dskysm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
double *alpha , const char *matdescra , const double *val , const MKL_INT *pntr , const
double *b , const MKL_INT *ldb , double *c , const MKL_INT *ldc );
```

```
void mkl_cskysm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT
*pntr , const MKL_Complex8 *b , const MKL_INT *ldb , MKL_Complex8 *c , const MKL_INT
*ldc );
```

```
void mkl_zskysm (const char *transa , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT
*pntr , const MKL_Complex16 *b , const MKL_INT *ldb , MKL_Complex16 *c , const MKL_INT
*ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_trsm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?skysm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the skyline storage format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(AT)*B,
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A*^T is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>transa</i>	Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $C := \alpha \cdot \text{inv}(A) \cdot B$, If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha \cdot \text{inv}(A^T) \cdot B$,
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>n</i>	Number of columns of the matrix <i>C</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .

NOTE

General matrices (*matdescra*[0]='G') is not supported.

<i>val</i>	Array containing the set of elements of the matrix <i>A</i> in the skyline profile form. If <i>matdescra</i> [2] = 'L', then <i>val</i> contains elements from the low triangle of the matrix <i>A</i> . If <i>matdescra</i> [2] = 'U', then <i>val</i> contains elements from the upper triangle of the matrix <i>A</i> . Refer to <i>values</i> array description in Skyline Storage Scheme for more details.
<i>pntr</i>	Array of length (<i>m</i> + 1) for lower triangle, and (<i>n</i> + 1) for upper triangle.

It contains the indices specifying the positions of the first element of the matrix A in each row (for the lower triangle) or column (for upper triangle) in the *val* array such that *val*[*pnt*[*i*] - 1] is the first element in row or column $i + 1$. Refer to *pointers* array description in [Skyline Storage Scheme](#) for more details.

<i>b</i>	Array, size <i>ldb</i> * <i>n</i> . On entry the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix B .
<i>ldb</i>	Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program.
<i>ldc</i>	Specifies the leading dimension of <i>c</i> as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Array, size <i>ldc</i> by <i>n</i> . The leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> contains the matrix C .
----------	---

mkl_?dnscsr

Convert a sparse matrix in uncompressed representation to the CSR format and vice versa (deprecated).

Syntax

```
void mkl_ddnscsr (const MKL_INT *job , const MKL_INT *m , const MKL_INT *n , double
*adns , const MKL_INT *lda , double *acsr , MKL_INT *ja , MKL_INT *ia , MKL_INT *info );
void mkl_sdnscsr (const MKL_INT *job , const MKL_INT *m , const MKL_INT *n , float
*adns , const MKL_INT *lda , float *acsr , MKL_INT *ja , MKL_INT *ia , MKL_INT *info );
void mkl_cdnscsr (const MKL_INT *job , const MKL_INT *m , const MKL_INT *n ,
MKL_Complex8 *adns , const MKL_INT *lda , MKL_Complex8 *acsr , MKL_INT *ja , MKL_INT
*ia , MKL_INT *info );
void mkl_zdnscsr (const MKL_INT *job , const MKL_INT *m , const MKL_INT *n ,
MKL_Complex16 *adns , const MKL_INT *lda , MKL_Complex16 *acsr , MKL_INT *ja , MKL_INT
*ia , MKL_INT *info );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use the [matrix manipulation routines](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix A between formats: stored as a rectangular array (dense representation) and stored using compressed sparse row (CSR) format (3-array variation).

Input Parameters

<i>job</i>	<p>Array, contains the following conversion parameters:</p> <ul style="list-style-type: none"> • <i>job</i>[0]: Conversion type. <ul style="list-style-type: none"> • If <i>job</i>[0]=0, the rectangular matrix <i>A</i> is converted to the CSR format; • if <i>job</i>[0]=1, the rectangular matrix <i>A</i> is restored from the CSR format. • <i>job</i>[1]: index base for the rectangular matrix <i>A</i>. <ul style="list-style-type: none"> • If <i>job</i>[1]=0, zero-based indexing for the rectangular matrix <i>A</i> is used; • if <i>job</i>[1]=1, one-based indexing for the rectangular matrix <i>A</i> is used. • <i>job</i>[2]: Index base for the matrix in CSR format. <ul style="list-style-type: none"> • If <i>job</i>[2]=0, zero-based indexing for the matrix in CSR format is used; • if <i>job</i>[2]=1, one-based indexing for the matrix in CSR format is used. • <i>job</i>[3]: Portion of matrix. <ul style="list-style-type: none"> • If <i>job</i>[3]=0, <i>adns</i> is a lower triangular part of matrix <i>A</i>; • If <i>job</i>[3]=1, <i>adns</i> is an upper triangular part of matrix <i>A</i>; • If <i>job</i>[3]=2, <i>adns</i> is a whole matrix <i>A</i>. • <i>job</i>[4]=<i>nzmax</i>: maximum number of the non-zero elements allowed if <i>job</i>[0]=0. • <i>job</i>[5]: job indicator for conversion to CSR format. <ul style="list-style-type: none"> • If <i>job</i>[5]=0, only array <i>ia</i> is generated for the output storage. • If <i>job</i>[5]>0, arrays <i>acsr</i>, <i>ia</i>, <i>ja</i> are generated for the output storage.
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>n</i>	Number of columns of the matrix <i>A</i> .
<i>adns</i>	<p>(input/output)</p> <p>If the conversion type is from uncompressed to CSR, on input <i>adns</i> contains an uncompressed (dense) representation of matrix <i>A</i>.</p>
<i>lda</i>	<p>Specifies the leading dimension of <i>adns</i> as declared in the calling (sub)program.</p> <p>For zero-based indexing of <i>A</i>, <i>lda</i> must be at least $\max(1, n)$.</p> <p>For one-based indexing of <i>A</i>, <i>lda</i> must be at least $\max(1, m)$.</p>
<i>acsr</i>	<p>(input/output)</p> <p>If conversion type is from CSR to uncompressed, on input <i>acsr</i> contains the non-zero elements of the matrix <i>A</i>. Its length is equal to the number of non-zero elements in the matrix <i>A</i>. Refer to values array description in Sparse Matrix Storage Formats for more details.</p>
<i>ja</i>	<p>(input/output). If conversion type is from CSR to uncompressed, on input for zero-based indexing of <i>A</i> <i>ja</i> contains the column indices plus one for each non-zero element of the matrix <i>A</i>. For one-based indexing of <i>A</i> <i>ja</i> contains the column indices for each non-zero element of the matrix <i>A</i>.</p>

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia

(input/output). Array of length $m + 1$.

If conversion type is from CSR to uncompressed, on input for zero-based indexing of *A* *ia* contains indices of elements in the array *acsr*, such that $ia[i] - 1$ is the index in the array *acsr* of the first non-zero element from the row *i*. For one-based indexing of *A* *ia* contains indices of elements in the array *acsr*, such that $ia[i]$ is the index in the array *acsr* of the first non-zero element from the row *i*.

The value of $ia[m] - ia[0]$ is equal to the number of non-zeros. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

Output Parameters

adns

If conversion type is from CSR to uncompressed, on output *adns* contains the uncompressed (dense) representation of matrix *A*.

acsr, ja, ia

If conversion type is from uncompressed to CSR, on output *acsr*, *ja*, and *ia* contain the compressed sparse row (CSR) format (3-array variation) of matrix *A* (see [Sparse Matrix Storage Formats](#) for a description of the storage format).

info

Integer info indicator only for restoring the matrix *A* from the CSR format.

If *info*=0, the execution is successful.

If *info*=*i*, the routine is interrupted processing the *i*-th row because there is no space in the arrays *acsr* and *ja* according to the value *nzmax*.

mkl_?csrcoo

Converts a sparse matrix in the CSR format to the coordinate format and vice versa (deprecated).

Syntax

```
void mkl_scsrcoo (const MKL_INT *job , const MKL_INT *n , float *acsr , MKL_INT *ja ,
MKL_INT *ia , MKL_INT *nnz , float *acoo , MKL_INT *rowind , MKL_INT *colind , MKL_INT
*info );
```

```
void mkl_dcsrcoo (const MKL_INT *job , const MKL_INT *n , double *acsr , MKL_INT *ja ,
MKL_INT *ia , MKL_INT *nnz , double *acoo , MKL_INT *rowind , MKL_INT *colind , MKL_INT
*info );
```

```
void mkl_ccsrcoo (const MKL_INT *job , const MKL_INT *n , MKL_Complex8 *acsr , MKL_INT
*ja , MKL_INT *ia , MKL_INT *nnz , MKL_Complex8 *acoo , MKL_INT *rowind , MKL_INT
*colind , MKL_INT *info );
```

```
void mkl_zcsrcoo (const MKL_INT *job , const MKL_INT *n , MKL_Complex16 *acsr , MKL_INT
*ja , MKL_INT *ia , MKL_INT *nnz , MKL_Complex16 *acoo , MKL_INT *rowind , MKL_INT
*colind , MKL_INT *info );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use the [matrix manipulation routines](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix *A* stored in the compressed sparse row (CSR) format (3-array variation) to coordinate format and vice versa.

Input Parameters

job Array, contains the following conversion parameters:

job[0]
 If *job*[0]=0, the matrix in the CSR format is converted to the coordinate format;
 if *job*[0]=1, the matrix in the coordinate format is converted to the CSR format.
 if *job*[0]=2, the matrix in the coordinate format is converted to the CSR format, and the column indices in CSR representation are sorted in the increasing order within each row.

job[1]
 If *job*[1]=0, zero-based indexing for the matrix in CSR format is used;
 if *job*[1]=1, one-based indexing for the matrix in CSR format is used.

job[2]
 If *job*[2]=0, zero-based indexing for the matrix in coordinate format is used;
 if *job*[2]=1, one-based indexing for the matrix in coordinate format is used.

job[4]
job[4]=*nzmax* - maximum number of the non-zero elements allowed if *job*[0]=0.

job[5] - job indicator.

For conversion to the coordinate format:
 If *job*[5]=1, only array *rowind* is filled in for the output storage.
 If *job*[5]=2, arrays *rowind*, *colind* are filled in for the output storage.
 If *job*[5]=3, all arrays *rowind*, *colind*, *acoo* are filled in for the output storage.

For conversion to the CSR format:
 If *job*[5]=0, all arrays *acsr*, *ja*, *ia* are filled in for the output storage.
 If *job*[5]=1, only array *ia* is filled in for the output storage.
 If *job*[5]=2, then it is assumed that the routine already has been called with the *job*[5]=1, and the user allocated the required space for storing the output arrays *acsr* and *ja*.

<i>n</i>	Dimension of the matrix A.
<i>nnz</i>	Specifies the number of non-zero elements of the matrix A for <i>job</i> [0]≠0. Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>acsr</i>	(input/output) Array containing non-zero elements of the matrix A. Its length is equal to the number of non-zero elements in the matrix A. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	(input/output). For <i>job</i> [1] = 1 (one-based indexing for the matrix in CSR format), array containing the column indices plus one for each non-zero element of the matrix A. For <i>job</i> [1] = 0 (zero-based indexing for the matrix in CSR format), array containing the column indices for each non-zero element of the matrix A. Its length is equal to the length of the array <i>acsr</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	(input/output). Array of length <i>n</i> + 1, containing indices of elements in the array <i>acsr</i> , such that <i>ia</i> [<i>i</i>] - <i>ia</i> [0] is the index in the array <i>acsr</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> [<i>n</i>] - <i>ia</i> [0] is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>acoo</i>	(input/output) Array containing non-zero elements of the matrix A. Its length is equal to the number of non-zero elements in the matrix A. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>rowind</i>	(input/output). Array of length <i>nnz</i> , contains the row indices for each non-zero element of the matrix A. Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	(input/output). Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix A. Refer to <i>columns</i> array description in Coordinate Format for more details.

Output Parameters

<i>nnz</i>	Returns the number of converted elements of the matrix A for <i>job</i> [0]=0.
<i>info</i>	Integer info indicator only for converting the matrix A from the CSR format. If <i>info</i> =0, the execution is successful. If <i>info</i> =1, the routine is interrupted because there is no space in the arrays <i>acoo</i> , <i>rowind</i> , <i>colind</i> according to the value <i>nzmax</i> .

`mkl_?csrbsr`

Converts a square sparse matrix in the CSR format to the BSR format and vice versa (deprecated).

Syntax

```
void mkl_scsrbsr (const MKL_INT *job , const MKL_INT *m , const MKL_INT *mblk , const
MKL_INT *ldabsr , float *acsr , MKL_INT *ja , MKL_INT *ia , float *absr , MKL_INT *jab ,
MKL_INT *iab , MKL_INT *info );

void mkl_dcsrbsr (const MKL_INT *job , const MKL_INT *m , const MKL_INT *mblk , const
MKL_INT *ldabsr , double *acsr , MKL_INT *ja , MKL_INT *ia , double *absr , MKL_INT
*jab , MKL_INT *iab , MKL_INT *info );

void mkl_ccsrbsr (const MKL_INT *job , const MKL_INT *m , const MKL_INT *mblk , const
MKL_INT *ldabsr , MKL_Complex8 *acsr , MKL_INT *ja , MKL_INT *ia , MKL_Complex8 *absr ,
MKL_INT *jab , MKL_INT *iab , MKL_INT *info );

void mkl_zcsrbsr (const MKL_INT *job , const MKL_INT *m , const MKL_INT *mblk , const
MKL_INT *ldabsr , MKL_Complex16 *acsr , MKL_INT *ja , MKL_INT *ia , MKL_Complex16
*absr , MKL_INT *jab , MKL_INT *iab , MKL_INT *info );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use the [matrix manipulation routines](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a square sparse matrix *A* stored in the compressed sparse row (CSR) format (3-array variation) to the block sparse row (BSR) format and vice versa.

Input Parameters

job

Array, contains the following conversion parameters:

job[0]

If *job*[0]=0, the matrix in the CSR format is converted to the BSR format;

if *job*[0]=1, the matrix in the BSR format is converted to the CSR format.

job[1]

If *job*[1]=0, zero-based indexing for the matrix in CSR format is used;

if *job*[1]=1, one-based indexing for the matrix in CSR format is used.

job[2]

If *job*[2]=0, zero-based indexing for the matrix in the BSR format is used;

if *job*[2]=1, one-based indexing for the matrix in the BSR format is used.

job[3] is only used for conversion to CSR format. By default, the converter saves the blocks without checking whether an element is zero or not. If *job*[3]=1, then the converter only saves non-zero elements in blocks.

job[5] - job indicator.

For conversion to the BSR format:

If *job*[5]=0, only arrays *jab*, *iab* are generated for the output storage.

If $job[5] > 0$, all output arrays *absr*, *jab*, and *iab* are filled in for the output storage.

If $job[5] = -1$, $iab[m] - iab[0]$ returns the number of non-zero blocks.

For conversion to the CSR format:

If $job[5] = 0$, only arrays *ja*, *ia* are generated for the output storage.

<i>m</i>	Actual row dimension of the matrix <i>A</i> for convert to the BSR format; block row dimension of the matrix <i>A</i> for convert to the CSR format.
<i>mblk</i>	Size of the block in the matrix <i>A</i> .
<i>ldabsr</i>	Leading dimension of the array <i>absr</i> as declared in the calling program. <i>ldabsr</i> must be greater than or equal to $mblk * mblk$.
<i>acsr</i>	(input/output) Array containing non-zero elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	(input/output). Array containing the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to the length of the array <i>acsr</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	(input/output). Array of length $m + 1$, containing indices of elements in the array <i>acsr</i> , such that $ia[I] - iab[0]$ is the index in the array <i>acsr</i> of the first non-zero element from the row <i>I</i> . The value of $ia[m] - iab[0]$ is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>absr</i>	(input/output) Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by $mblk * mblk$. Refer to <i>values</i> array description in BSR Format for more details.
<i>jab</i>	(input/output). Array containing the column indices for each non-zero block of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks of the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>iab</i>	(input/output). Array of length $(m + 1)$, containing indices of blocks in the array <i>absr</i> , such that $iab[i] - iab[0]$ is the index in the array <i>absr</i> of the first non-zero element from the <i>i</i> -th row. The value of $iab[m]$ is equal to the number of non-zero blocks. Refer to <i>rowIndex</i> array description in BSR Format for more details.

Output Parameters

<i>info</i>	Integer info indicator only for converting the matrix <i>A</i> from the CSR format. If $info = 0$, the execution is successful. If $info = 1$, it means that <i>mblk</i> is equal to 0.
-------------	---

If *info*=2, it means that *ldabsr* is less than *mb1k*mb1k* and there is no space for all blocks.

mkl_?csrsc

Converts a square sparse matrix in the CSR format to the CSC format and vice versa (deprecated).

Syntax

```
void mkl_dcsrsc (const MKL_INT *job , const MKL_INT *n , double *acsr , MKL_INT *ja ,
MKL_INT *ia , double *acsc , MKL_INT *jal , MKL_INT *ial , MKL_INT *info );

void mkl_scsrsc (const MKL_INT *job , const MKL_INT *n , float *acsr , MKL_INT *ja ,
MKL_INT *ia , float *acsc , MKL_INT *jal , MKL_INT *ial , MKL_INT *info );

void mkl_ccsrsc (const MKL_INT *job , const MKL_INT *n , MKL_Complex8 *acsr , MKL_INT
*ja , MKL_INT *ia , MKL_Complex8 *acsc , MKL_INT *jal , MKL_INT *ial , MKL_INT *info );

void mkl_zcsrsc (const MKL_INT *job , const MKL_INT *n , MKL_Complex16 *acsr , MKL_INT
*ja , MKL_INT *ia , MKL_Complex16 *acsc , MKL_INT *jal , MKL_INT *ial , MKL_INT *info );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use the [matrix manipulation routines](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a square sparse matrix *A* stored in the compressed sparse row (CSR) format (3-array variation) to the compressed sparse column (CSC) format and vice versa.

Input Parameters

job

Array, contains the following conversion parameters:

job[0]

If *job*[0]=0, the matrix in the CSR format is converted to the CSC format;

if *job*[0]=1, the matrix in the CSC format is converted to the CSR format.

job[1]

If *job*[1]=0, zero-based indexing for the matrix in CSR format is used;

if *job*[1]=1, one-based indexing for the matrix in CSR format is used.

job[2]

If *job*[2]=0, zero-based indexing for the matrix in the CSC format is used;

if *job*[2]=1, one-based indexing for the matrix in the CSC format is used.

job[5] - job indicator.

For conversion to the CSC format:

If *job*[5]=0, only arrays *jal*, *ial* are filled in for the output storage.

If $job[5] \neq 0$, all output arrays *acsc*, *jal*, and *ial* are filled in for the output storage.

For conversion to the CSR format:

If $job[5] = 0$, only arrays *ja*, *ia* are filled in for the output storage.

If $job[5] \neq 0$, all output arrays *acsr*, *ja*, and *ia* are filled in for the output storage.

<i>m</i>	Dimension of the square matrix <i>A</i> .
<i>acsr</i>	(input/output) Array containing non-zero elements of the square matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	(input/output). Array containing the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to the length of the array <i>acsr</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	(input/output). Array of length $m + 1$, containing indices of elements in the array <i>acsr</i> , such that $ia[i] - ia[0]$ is the index in the array <i>acsr</i> of the first non-zero element from the row <i>i</i> . The value of $ia[m] - ia[0]$ is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>acsc</i>	(input/output) Array containing non-zero elements of the square matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>jal</i>	(input/output). Array containing the row indices for each non-zero element of the matrix <i>A</i> . Its length is equal to the length of the array <i>acsc</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>ial</i>	(input/output). Array of length $m + 1$, containing indices of elements in the array <i>acsc</i> , such that $ial[i] - ial[0]$ is the index in the array <i>acsc</i> of the first non-zero element from the column <i>i</i> . The value of $ial[m] - ial[0]$ is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.

Output Parameters

<i>info</i>	This parameter is not used now.
-------------	---------------------------------

mkl_?csrdia

Converts a sparse matrix in the CSR format to the diagonal format and vice versa (deprecated).

Syntax

```
void mkl_dcsrdia (const MKL_INT *job , const MKL_INT *n , double *acsr , MKL_INT *ja ,
MKL_INT *ia , double *adia , const MKL_INT *ndiag , MKL_INT *distance , MKL_INT
*idiag , double *acsr_rem , MKL_INT *ja_rem , MKL_INT *ia_rem , MKL_INT *info );

void mkl_scsrdia (const MKL_INT *job , const MKL_INT *n , float *acsr , MKL_INT *ja ,
MKL_INT *ia , float *adia , const MKL_INT *ndiag , MKL_INT *distance , MKL_INT *idiag ,
float *acsr_rem , MKL_INT *ja_rem , MKL_INT *ia_rem , MKL_INT *info );

void mkl_ccsrdia (const MKL_INT *job , const MKL_INT *n , MKL_Complex8 *acsr , MKL_INT
*ja , MKL_INT *ia , MKL_Complex8 *adia , const MKL_INT *ndiag , MKL_INT *distance ,
MKL_INT *idiag , MKL_Complex8 *acsr_rem , MKL_INT *ja_rem , MKL_INT *ia_rem , MKL_INT
*info );

void mkl_zcsrdia (const MKL_INT *job , const MKL_INT *n , MKL_Complex16 *acsr , MKL_INT
*ja , MKL_INT *ia , MKL_Complex16 *adia , const MKL_INT *ndiag , MKL_INT *distance ,
MKL_INT *idiag , MKL_Complex16 *acsr_rem , MKL_INT *ja_rem , MKL_INT *ia_rem , MKL_INT
*info );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use the [matrix manipulation routines](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix *A* stored in the compressed sparse row (CSR) format (3-array variation) to the diagonal format and vice versa.

Input Parameters

job

Array, contains the following conversion parameters:

job[0]

If *job*[0]=0, the matrix in the CSR format is converted to the diagonal format;

if *job*[0]=1, the matrix in the diagonal format is converted to the CSR format.

job[1]

If *job*[1]=0, zero-based indexing for the matrix in CSR format is used;

if *job*[1]=1, one-based indexing for the matrix in CSR format is used.

job[2]

If *job*[2]=0, zero-based indexing for the matrix in the diagonal format is used;

if *job*[2]=1, one-based indexing for the matrix in the diagonal format is used.

job[5] - job indicator.

For conversion to the diagonal format:

If `job[5]=0`, diagonals are not selected internally, and `acsr_rem`, `ja_rem`, `ia_rem` are not filled in for the output storage.

If `job[5]=1`, diagonals are not selected internally, and `acsr_rem`, `ja_rem`, `ia_rem` are filled in for the output storage.

If `job[5]=10`, diagonals are selected internally, and `acsr_rem`, `ja_rem`, `ia_rem` are not filled in for the output storage.

If `job[5]=11`, diagonals are selected internally, and `csr_rem`, `ja_rem`, `ia_rem` are filled in for the output storage.

For conversion to the CSR format:

If `job[5]=0`, each entry in the array `adia` is checked whether it is zero. Zero entries are not included in the array `acsr`.

If `job[5]≠0`, each entry in the array `adia` is not checked whether it is zero.

`m`

Dimension of the matrix `A`.

`acsr`

(input/output)

Array containing non-zero elements of the matrix `A`. Its length is equal to the number of non-zero elements in the matrix `A`. Refer to `values` array description in [Sparse Matrix Storage Formats](#) for more details.

`ja`

(input/output). Array containing the column indices for each non-zero element of the matrix `A`.

Its length is equal to the length of the array `acsr`. Refer to `columns` array description in [Sparse Matrix Storage Formats](#) for more details.

`ia`

(input/output). Array of length $m + 1$, containing indices of elements in the array `acsr`, such that `ia[i] - ia[0]` is the index in the array `acsr` of the first non-zero element from the row `i`. The value of `ia[m] - ia[0]` is equal to the number of non-zeros. Refer to `rowIndex` array description in [Sparse Matrix Storage Formats](#) for more details.

`adia`

(input/output)

Array of size $(ndiag * idiag)$ containing diagonals of the matrix `A`.

The key point of the storage is that each element in the array `adia` retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom.

`ndiag`

Specifies the leading dimension of the array `adia` as declared in the calling (sub)program, must be at least $\max(1, m)$.

`distance`

Array of length `idiag`, containing the distances between the main diagonal and each non-zero diagonal to be extracted. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

`idiag`

Number of diagonals to be extracted. For conversion to diagonal format on return this parameter may be modified.

`acsr_rem, ja_rem, ia_rem`

Remainder of the matrix in the CSR format if it is needed for conversion to the diagonal format.

Output Parameters

info

This parameter is not used now.

mkl_?csrsky

Converts a sparse matrix in CSR format to the skyline format and vice versa (deprecated).

Syntax

```
void mkl_dcsrsky (const MKL_INT *job , const MKL_INT *m , double *acsr , MKL_INT *ja ,
MKL_INT *ia , double *asky , MKL_INT *pointers , MKL_INT *info );

void mkl_scsrsky (const MKL_INT *job , const MKL_INT *m , float *acsr , MKL_INT *ja ,
MKL_INT *ia , float *asky , MKL_INT *pointers , MKL_INT *info );

void mkl_ccsrsky (const MKL_INT *job , const MKL_INT *m , MKL_Complex8 *acsr , MKL_INT
*ja , MKL_INT *ia , MKL_Complex8 *asky , MKL_INT *pointers , MKL_INT *info );

void mkl_zcsrsky (const MKL_INT *job , const MKL_INT *m , MKL_Complex16 *acsr , MKL_INT
*ja , MKL_INT *ia , MKL_Complex16 *asky , MKL_INT *pointers , MKL_INT *info );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use the [matrix manipulation routines](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix *A* stored in the compressed sparse row (CSR) format (3-array variation) to the skyline format and vice versa.

Input Parameters

job

Array, contains the following conversion parameters:

job[0]

If *job*[0]=0, the matrix in the CSR format is converted to the skyline format;

if *job*[0]=1, the matrix in the skyline format is converted to the CSR format.

job[1]

If *job*[1]=0, zero-based indexing for the matrix in CSR format is used;

if *job*[1]=1, one-based indexing for the matrix in CSR format is used.

job[2]

If *job*[2]=0, zero-based indexing for the matrix in the skyline format is used;

if *job*[2]=1, one-based indexing for the matrix in the skyline format is used.

job[3]

For conversion to the skyline format:

If $job[3]=0$, the upper part of the matrix A in the CSR format is converted.

If $job[3]=1$, the lower part of the matrix A in the CSR format is converted.

For conversion to the CSR format:

If $job[3]=0$, the matrix is converted to the upper part of the matrix A in the CSR format.

If $job[3]=1$, the matrix is converted to the lower part of the matrix A in the CSR format.

$job[4]$

$job[4]=nzmax$ - maximum number of the non-zero elements of the matrix A if $job[0]=0$.

$job[5]$ - job indicator.

Only for conversion to the skyline format:

If $job[5]=0$, only arrays *pointers* is filled in for the output storage.

If $job[5]=1$, all output arrays *asky* and *pointers* are filled in for the output storage.

m

Dimension of the matrix A .

acsr

(input/output)

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja

(input/output). Array containing the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia

(input/output). Array of length $m + 1$, containing indices of elements in the array *acsr*, such that $ia[i] - ia[0]$ is the index in the array *acsr* of the first non-zero element from the row i . The value of $ia[m] - ia[0]$ is equal to the number of non-zeros. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

asky

(input/output)

Array, for a lower triangular part of A it contains the set of elements from each row starting from the first non-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets. Refer to *values* array description in [Skyline Storage Format](#) for more details.

pointers

(input/output).

Array with dimension $(m+1)$, where m is number of rows for lower triangle (columns for upper triangle), $pointers[i-1] - pointers[0]$ gives the index of element in the array *asky* that is first non-zero element in row

(column) i . The value of `pointers[m]` is set to `nnz + pointers[0]`, where `nnz` is the number of elements in the array `asky`. Refer to `pointers` array description in [Skyline Storage Format](#) for more details

Output Parameters

`info`

Integer info indicator only for converting the matrix *A* from the CSR format.

If `info=0`, the execution is successful.

If `info=1`, the routine is interrupted because there is no space in the array `asky` according to the value `nzmax`.

`mkl_?csradd`

Computes the sum of two matrices stored in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
void mkl_dcsradd (const char *trans , const MKL_INT *request , const MKL_INT *sort ,
const MKL_INT *m , const MKL_INT *n , double *a , MKL_INT *ja , MKL_INT *ia , const
double *beta , double *b , MKL_INT *jb , MKL_INT *ib , double *c , MKL_INT *jc , MKL_INT
*ic , const MKL_INT *nzmax , MKL_INT *info );
```

```
void mkl_scsradd (const char *trans , const MKL_INT *request , const MKL_INT *sort ,
const MKL_INT *m , const MKL_INT *n , float *a , MKL_INT *ja , MKL_INT *ia , const
float *beta , float *b , MKL_INT *jb , MKL_INT *ib , float *c , MKL_INT *jc , MKL_INT
*ic , const MKL_INT *nzmax , MKL_INT *info );
```

```
void mkl_ccsradd (const char *trans , const MKL_INT *request , const MKL_INT *sort ,
const MKL_INT *m , const MKL_INT *n , MKL_Complex8 *a , MKL_INT *ja , MKL_INT *ia ,
const MKL_Complex8 *beta , MKL_Complex8 *b , MKL_INT *jb , MKL_INT *ib , MKL_Complex8
*c , MKL_INT *jc , MKL_INT *ic , const MKL_INT *nzmax , MKL_INT *info );
```

```
void mkl_zcsradd (const char *trans , const MKL_INT *request , const MKL_INT *sort ,
const MKL_INT *m , const MKL_INT *n , MKL_Complex16 *a , MKL_INT *ja , MKL_INT *ia ,
const MKL_Complex16 *beta , MKL_Complex16 *b , MKL_INT *jb , MKL_INT *ib ,
MKL_Complex16 *c , MKL_INT *jc , MKL_INT *ic , const MKL_INT *nzmax , MKL_INT *info );
```

Include Files

- `mkl.h`

Description

This routine is deprecated. Use `mkl_sparse_?_add` from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csradd` routine performs a matrix-matrix operation defined as

$$C := A + \text{beta} * \text{op}(B)$$

where:

A, *B*, *C* are the sparse matrices in the CSR format (3-array variation).

`op(B)` is one of `op(B) = B`, or `op(B) = BT`, or `op(B) = BH`

beta is a scalar.

The routine works correctly if and only if the column indices in sparse matrix representations of matrices A and B are arranged in the increasing order for each row. If not, use the parameter `sort` (see below) to reorder column indices and the corresponding elements of the input matrices.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<code>trans</code>	<p>Specifies the operation.</p> <p>If <code>trans = 'N' or 'n'</code>, then $C := A + \text{beta} * B$</p> <p>If <code>trans = 'T' or 't'</code>, then $C := A + \text{beta} * B^T$</p> <p>If <code>trans = 'C' or 'c'</code>, then $C := A + \text{beta} * B^H$.</p>
<code>request</code>	<p>If <code>request=0</code>, the routine performs addition. The memory for the output arrays <code>ic</code>, <code>jc</code>, <code>c</code> must be allocated beforehand.</p> <p>If <code>request=1</code>, the routine only computes the values of the array <code>ic</code> of length $m + 1$. The memory for the <code>ic</code> array must be allocated beforehand. On exit the value <code>ic[m] - 1</code> is the actual number of the elements in the arrays <code>c</code> and <code>jc</code>.</p> <p>If <code>request=2</code>, after the routine is called previously with the parameter <code>request=1</code> and after the output arrays <code>jc</code> and <code>c</code> are allocated in the calling program with length at least <code>ic[m] - 1</code>, the routine performs addition.</p>
<code>sort</code>	<p>Specifies the type of reordering. If this parameter is not set (default), the routine does not perform reordering.</p> <p>If <code>sort=1</code>, the routine arranges the column indices <code>ja</code> for each row in the increasing order and reorders the corresponding values of the matrix A in the array <code>a</code>.</p> <p>If <code>sort=2</code>, the routine arranges the column indices <code>jb</code> for each row in the increasing order and reorders the corresponding values of the matrix B in the array <code>b</code>.</p> <p>If <code>sort=3</code>, the routine performs reordering for both input matrices A and B.</p>
<code>m</code>	Number of rows of the matrix A .
<code>n</code>	Number of columns of the matrix A .
<code>a</code>	Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to <code>values</code> array description in Sparse Matrix Storage Formats for more details.
<code>ja</code>	<p>Array containing the column indices plus one for each non-zero element of the matrix A. For each row the column indices must be arranged in the increasing order.</p> <p>The length of this array is equal to the length of the array <code>a</code>. Refer to <code>columns</code> array description in Sparse Matrix Storage Formats for more details.</p>

<i>ia</i>	Array of length $m + 1$, containing indices of elements in the array <i>a</i> , such that $ia[i] - ia[0]$ is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element $ia[m]$ is equal to the number of non-zero elements of the matrix <i>A</i> plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>beta</i>	Specifies the scalar <i>beta</i> .
<i>b</i>	Array containing non-zero elements of the matrix <i>B</i> . Its length is equal to the number of non-zero elements in the matrix <i>B</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>jb</i>	Array containing the column indices plus one for each non-zero element of the matrix <i>B</i> . For each row the column indices must be arranged in the increasing order. The length of this array is equal to the length of the array <i>b</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>ib</i>	Array of length $m + 1$ when <i>trans</i> = 'N' or 'n', or $n + 1$ otherwise. This array contains indices of elements in the array <i>b</i> , such that $ib[i] - ib[0]$ is the index in the array <i>b</i> of the first non-zero element from the row <i>i</i> . The value of the last element $ib[m]$ or $ib[n]$ is equal to the number of non-zero elements of the matrix <i>B</i> plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>nzmax</i>	The length of the arrays <i>c</i> and <i>jc</i> . This parameter is used only if <i>request</i> =0. The routine stops calculation if the number of elements in the result matrix <i>C</i> exceeds the specified value of <i>nzmax</i> .

Output Parameters

<i>c</i>	Array containing non-zero elements of the result matrix <i>C</i> . Its length is equal to the number of non-zero elements in the matrix <i>C</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>jc</i>	Array containing the column indices plus one for each non-zero element of the matrix <i>C</i> . The length of this array is equal to the length of the array <i>c</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>ic</i>	Array of length $m + 1$, containing indices of elements in the array <i>c</i> , such that $ic[i] - ic[0]$ is the index in the array <i>c</i> of the first non-zero element from the row <i>i</i> . The value of the last element $ic[m]$ is equal to the number of non-zero elements of the matrix <i>C</i> plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>info</i>	If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>I</i> >0, the routine stops calculation in the <i>I</i> -th row of the matrix <i>C</i> because number of elements in <i>C</i> exceeds <i>nzmax</i> .

If *info*=-1, the routine calculates only the size of the arrays *c* and *jc* and returns this value plus 1 as the last element of the array *ic*.

mkl_?csrmultcsr

Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

```
void mkl_dcsmultcsr (const char *trans , const MKL_INT *request , const MKL_INT
*sort , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , double *a , MKL_INT
*ja , MKL_INT *ia , double *b , MKL_INT *jb , MKL_INT *ib , double *c , MKL_INT *jc ,
MKL_INT *ic , const MKL_INT *nzmax , MKL_INT *info );

void mkl_scsmultcsr (const char *trans , const MKL_INT *request , const MKL_INT
*sort , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , float *a , MKL_INT
*ja , MKL_INT *ia , float *b , MKL_INT *jb , MKL_INT *ib , float *c , MKL_INT *jc ,
MKL_INT *ic , const MKL_INT *nzmax , MKL_INT *info );

void mkl_ccsmultcsr (const char *trans , const MKL_INT *request , const MKL_INT
*sort , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , MKL_Complex8 *a ,
MKL_INT *ja , MKL_INT *ia , MKL_Complex8 *b , MKL_INT *jb , MKL_INT *ib , MKL_Complex8
*c , MKL_INT *jc , MKL_INT *ic , const MKL_INT *nzmax , MKL_INT *info );

void mkl_zcsmultcsr (const char *trans , const MKL_INT *request , const MKL_INT
*sort , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , MKL_Complex16 *a ,
MKL_INT *ja , MKL_INT *ia , MKL_Complex16 *b , MKL_INT *jb , MKL_INT *ib ,
MKL_Complex16 *c , MKL_INT *jc , MKL_INT *ic , const MKL_INT *nzmax , MKL_INT *info );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_spmv](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrmultcsr` routine performs a matrix-matrix operation defined as

$$C := \text{op}(A) * B$$

where:

A, *B*, *C* are the sparse matrices in the CSR format (3-array variation);

$\text{op}(A)$ is one of $\text{op}(A) = A$, or $\text{op}(A) = A^T$, or $\text{op}(A) = A^H$.

You can use the parameter *sort* to perform or not perform reordering of non-zero entries in input and output sparse matrices. The purpose of reordering is to rearrange non-zero entries in compressed sparse row matrix so that column indices in compressed sparse representation are sorted in the increasing order for each row.

The following table shows correspondence between the value of the parameter *sort* and the type of reordering performed by this routine for each sparse matrix involved:

Value of the parameter <i>sort</i>	Reordering of <i>A</i> (arrays <i>a, ja, ia</i>)	Reordering of <i>B</i> (arrays <i>b, ja, ib</i>)	Reordering of <i>C</i> (arrays <i>c, jc, ic</i>)
1	yes	no	yes
2	no	yes	yes
3	yes	yes	yes
4	yes	no	no
5	no	yes	no
6	yes	yes	no
7	no	no	no
arbitrary value not equal to 1, 2,..., 7	no	no	yes

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

<i>trans</i>	<p>Specifies the operation.</p> <p>If <i>trans</i> = 'N' or 'n', then $C := A * B$</p> <p>If <i>trans</i> = 'T' or 't' or 'C' or 'c', then $C := A^T * B$.</p>
<i>request</i>	<p>If <i>request</i>=0, the routine performs multiplication, the memory for the output arrays <i>ic</i>, <i>jc</i>, <i>c</i> must be allocated beforehand.</p> <p>If <i>request</i>=1, the routine computes only values of the array <i>ic</i> of length <i>m</i> + 1, the memory for this array must be allocated beforehand. On exit the value <i>ic</i>[<i>m</i>] - 1 is the actual number of the elements in the arrays <i>c</i> and <i>jc</i>.</p> <p>If <i>request</i>=2, the routine has been called previously with the parameter <i>request</i>=1, the output arrays <i>jc</i> and <i>c</i> are allocated in the calling program and they are of the length <i>ic</i>[<i>m</i>] - 1 at least.</p>
<i>sort</i>	Specifies whether the routine performs reordering of non-zeros entries in input and/or output sparse matrices (see table above).
<i>m</i>	Number of rows of the matrix <i>A</i> .
<i>n</i>	Number of columns of the matrix <i>A</i> .
<i>k</i>	Number of columns of the matrix <i>B</i> .
<i>a</i>	Array containing non-zero elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	<p>Array containing the column indices plus one for each non-zero element of the matrix <i>A</i>. For each row the column indices must be arranged in the increasing order.</p> <p>The length of this array is equal to the length of the array <i>a</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>

<i>ia</i>	<p>Array of length $m + 1$.</p> <p>This array contains indices of elements in the array <i>a</i>, such that $ia[i] - ia[0]$ is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i>. The value of the last element $ia[m]$ is equal to the number of non-zero elements of the matrix <i>A</i> plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>b</i>	<p>Array containing non-zero elements of the matrix <i>B</i>. Its length is equal to the number of non-zero elements in the matrix <i>B</i>. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>jb</i>	<p>Array containing the column indices plus one for each non-zero element of the matrix <i>B</i>. For each row the column indices must be arranged in the increasing order.</p> <p>The length of this array is equal to the length of the array <i>b</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ib</i>	<p>Array of length $n + 1$ when <i>trans</i> = 'N' or 'n', or $m + 1$ otherwise.</p> <p>This array contains indices of elements in the array <i>b</i>, such that $ib[i] - ib[0]$ is the index in the array <i>b</i> of the first non-zero element from the row <i>i</i>. The value of the last element $ib[n]$ or $ib[m]$ is equal to the number of non-zero elements of the matrix <i>B</i> plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>nzmax</i>	<p>The length of the arrays <i>c</i> and <i>jc</i>.</p> <p>This parameter is used only if <i>request</i>=0. The routine stops calculation if the number of elements in the result matrix <i>C</i> exceeds the specified value of <i>nzmax</i>.</p>

Output Parameters

<i>c</i>	<p>Array containing non-zero elements of the result matrix <i>C</i>. Its length is equal to the number of non-zero elements in the matrix <i>C</i>. Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>jc</i>	<p>Array containing the column indices plus one for each non-zero element of the matrix <i>C</i>.</p> <p>The length of this array is equal to the length of the array <i>c</i>. Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>ic</i>	<p>Array of length $m + 1$ when <i>trans</i> = 'N' or 'n', or $n + 1$ otherwise.</p> <p>This array contains indices of elements in the array <i>c</i>, such that $ic[i] - ic[0]$ is the index in the array <i>c</i> of the first non-zero element from the row <i>i</i>. The value of the last element $ic[m]$ or $ic[n]$ is equal to the number of non-zero elements of the matrix <i>C</i> plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.</p>
<i>info</i>	<p>If <i>info</i>=0, the execution is successful.</p> <p>If <i>info</i>=<i>I</i>>0, the routine stops calculation in the <i>I</i>-th row of the matrix <i>C</i> because number of elements in <i>C</i> exceeds <i>nzmax</i>.</p>

If *info*=-1, the routine calculates only the size of the arrays *c* and *jc* and returns this value plus 1 as the last element of the array *ic*.

mkl_?csrmultd

Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing. The result is stored in the dense matrix (deprecated).

Syntax

```
void mkl_dcsmultd (const char *trans , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , double *a , MKL_INT *ja , MKL_INT *ia , double *b , MKL_INT *jb , MKL_INT
*ib , double *c , MKL_INT *ldc );
```

```
void mkl_scsmultd (const char *trans , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , float *a , MKL_INT *ja , MKL_INT *ia , float *b , MKL_INT *jb , MKL_INT
*ib , float *c , MKL_INT *ldc );
```

```
void mkl_ccsmultd (const char *trans , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , MKL_Complex8 *a , MKL_INT *ja , MKL_INT *ia , MKL_Complex8 *b , MKL_INT
*jb , MKL_INT *ib , MKL_Complex8 *c , MKL_INT *ldc );
```

```
void mkl_zcsmultd (const char *trans , const MKL_INT *m , const MKL_INT *n , const
MKL_INT *k , MKL_Complex16 *a , MKL_INT *ja , MKL_INT *ia , MKL_Complex16 *b , MKL_INT
*jb , MKL_INT *ib , MKL_Complex16 *c , MKL_INT *ldc );
```

Include Files

- mkl.h

Description

This routine is deprecated. Use [mkl_sparse_?_spmm](#) from the Intel® oneAPI Math Kernel Library (oneMKL) Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrmultd` routine performs a matrix-matrix operation defined as

$$C := \text{op}(A) * B$$

where:

A, *B* are the sparse matrices in the CSR format (3-array variation), *C* is dense matrix;

$\text{op}(A)$ is one of $\text{op}(A) = A$, or $\text{op}(A) = A^T$, or $\text{op}(A) = A^H$.

The routine works correctly if and only if the column indices in sparse matrix representations of matrices *A* and *B* are arranged in the increasing order for each row.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

trans

Specifies the operation.

If *trans* = 'N' or 'n', then $C := A * B$

If $trans = 'T'$ or $'t'$ or $'C'$ or $'c'$, then $C := A^T * B$.

m

Number of rows of the matrix A .

n

Number of columns of the matrix A .

k

Number of columns of the matrix B .

a

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja

Array containing the column indices plus one for each non-zero element of the matrix A . For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array a . Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia

Array of length $m + 1$ when $trans = 'N'$ or $'n'$, or $n + 1$ otherwise.

This array contains indices of elements in the array a , such that $ia[i] - ia[0]$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia[m]$ or $ia[n]$ is equal to the number of non-zero elements of the matrix A plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

b

Array containing non-zero elements of the matrix B . Its length is equal to the number of non-zero elements in the matrix B . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

jb

Array containing the column indices plus one for each non-zero element of the matrix B . For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array b . Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ib

Array of length $m + 1$.

This array contains indices of elements in the array b , such that $ib[i] - ib[0]$ is the index in the array b of the first non-zero element from the row i . The value of the last element $ib[m]$ is equal to the number of non-zero elements of the matrix B plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

Output Parameters

c

Array containing non-zero elements of the result matrix C .

ldc

Specifies the leading dimension of the dense matrix C as declared in the calling (sub)program. Must be at least $\max(m, 1)$ when $trans = 'N'$ or $'n'$, or $\max(1, n)$ otherwise.

Sparse QR Routines

Sparse QR routines and their data types

Routine or function group	Data types	Description
mkl_sparse_set_qr_hint		Enables a pivot strategy for an ill-conditioned matrix.
mkl_sparse_?_qr	s,d	Calculates the solution of a sparse system of linear equations using QR factorization.
mkl_sparse_qr_reorder		Performs reordering and symbolic analysis of the matrix A .
mkl_sparse_?_qr_factorize	s,d	Performs numerical factorization of the matrix A .
mkl_sparse_?_qr_solve	s,d	Solves the system $A*x = b$ using QR factorization of the matrix A .
mkl_sparse_?_qr_qmult	s,d	Performs $x := Q^{(-1)} * b$.
mkl_sparse_?_qr_ksolve	s,d	Performs $x := R^{(-1)} * b$.

NOTE The underdetermined systems of equations are not supported. The number of columns should be less or equal to the number or rows.

For more information about the workflow of sparse QR functionality, refer to [oneMKL Sparse QR solver. Multifrontal Sparse QR Factorization Method for Solving a Sparse System of Linear Equations](#).

[mkl_sparse_set_qr_hint](#)

Define the pivot strategy for further calls of [mkl_sparse_?_qr](#).

Syntax

```
sparse_status_t mkl_sparse_set_qr_hint (sparse_matrix_t A, sparse_qr_hint_t hint);
```

Include Files

- `mkl_sparse_qr.h`

Description

You can use this routine to enable a pivot strategy in the case of an ill-conditioned matrix.

Input Parameters

A Handle containing a sparse matrix in an internal data structure.

hint Value specifying whether to use pivoting.

NOTE The only value currently supported is `SPARSE_QR_WITH_PIVOTS`, which enables the use of a pivot strategy for an ill-conditioned matrix.

Return Values

`SPARSE_STATUS_SUCCESS` The operation was successful.

`SPARSE_STATUS_NOT_INITIALIZED` The routine encountered an empty handle or matrix array.

`SPARSE_STATUS_ALLOC_FAILED` Internal memory allocation failed.

`SPARSE_STATUS_INVALID_VALUE` The input parameters contain an invalid value.

`SPARSE_STATUS_EXECUTION_FAILED` Execution failed.

`SPARSE_STATUS_INTERNAL_ERROR` An error in algorithm implementation occurred.

`SPARSE_STATUS_NOT_SUPPORTED` The requested operation is not supported.

mkl_sparse?_qr

Computes the QR decomposition for the matrix of a sparse linear system and calculates the solution.

Syntax

```
sparse_status_t mkl_sparse_d_qr ( sparse_operation_t operation, sparse_matrix_t A,
struct matrix_descr descr, sparse_layout_t layout, MKL_INT columns, double *x, MKL_INT
ldx, const double *b, MKL_INT ldb );
```

```
sparse_status_t mkl_sparse_s_qr ( sparse_operation_t operation, sparse_matrix_t A,
struct matrix_descr descr, sparse_layout_t layout, MKL_INT columns, float *x, MKL_INT
ldx, const float *b, MKL_INT ldb );
```

Include Files

- `mkl_sparse_qr.h`

Description

The `mkl_sparse?_qr` routine computes the QR decomposition for the matrix of a sparse linear system $A*x = b$, so that $A = Q*R$ where Q is the orthogonal matrix and R is upper triangular, and calculates the solution.

NOTE

Currently, `mkl_sparse?_qr` supports only square and overdetermined systems. For underdetermined systems you can manually transpose the system matrix and use QR decomposition for A^T to get the minimum-norm solution for the original underdetermined system.

NOTE Currently, `mkl_sparse?_qr` supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.

Input Parameters

`operation` Specifies the operation to perform.

NOTE Currently, the only supported value is `SPARSE_OPERATION_NON_TRANSPOSE` (non-transpose case; that is, $A^*x = b$ is solved).

A Handle containing a sparse matrix in an internal data structure.

descr Structure specifying sparse matrix properties. Only the parameters listed here are currently supported.

type Specifies the type of sparse matrix.

NOTE Currently, the only supported value is `SPARSE_MATRIX_TYPE_GENERAL` (the matrix is processed as-is).

layout Describes the storage scheme for the dense matrix:

<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Storage of elements uses column-major layout.
<code>SPARSE_LAYOUT_ROW_MAJOR</code>	Storage of elements uses row-major layout.

x Array with a size of at least `rows*cols`:

	<i>layout</i> = <code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	<i>layout</i> = <code>SPARSE_LAYOUT_ROW_MAJOR</code>
<i>rows</i> (number of rows in <i>x</i>)	<i>ldx</i>	Number of columns in <i>A</i>
<i>cols</i> (number of columns in <i>x</i>)	<i>columns</i>	<i>ldx</i>

columns Number of columns in matrix *b*.

ldx Specifies the leading dimension of matrix *x*.

b Array with a size of at least `rows*cols`:

	<i>layout</i> = <code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	<i>layout</i> = <code>SPARSE_LAYOUT_ROW_MAJOR</code>
<i>rows</i> (number of rows in <i>b</i>)	<i>ldb</i>	Number of columns in <i>A</i>
<i>cols</i> (number of columns in <i>b</i>)	<i>columns</i>	<i>ldb</i>

ldb Specifies the leading dimension of matrix *b*.

Output Parameters

`x` Overwritten by the updated matrix `y`.

Return Values

`SPARSE_STATUS_SUCCESS` The operation was successful.

`SPARSE_STATUS_NOT_INITIALIZED` The routine encountered an empty handle or matrix array.

`SPARSE_STATUS_ALLOC_FAILED` Internal memory allocation failed.

`SPARSE_STATUS_INVALID_VALUE` The input parameters contain an invalid value.

`SPARSE_STATUS_EXECUTION_FAILED` Execution failed.

`SPARSE_STATUS_INTERNAL_ERROR` An error in algorithm implementation occurred.

`SPARSE_STATUS_NOT_SUPPORTED` The requested operation is not supported.

`mkl_sparse_qr_reorder`

Reordering step of SPARSE QR solver.

Syntax

```
sparse_status_t mkl_sparse_qr_reorder (sparse_matrix_t A, struct matrix_descr descr);
```

Include Files

- `mkl_sparse_qr.h`

Description

The `mkl_sparse_qr_reorder` routine performs ordering and symbolic analysis of matrix `A`.

NOTE Currently, `mkl_sparse_qr_reorder` supports only general structure and CSR format for the input matrix.

Input Parameters

`A` Handle containing a sparse matrix in an internal data structure.

`descr` Structure specifying sparse matrix properties. Only the parameters listed here are currently supported.

Return Values

`SPARSE_STATUS_SUCCESS` The operation was successful.

`SPARSE_STATUS_NOT_INITIALIZED` The routine encountered an empty handle or matrix array.

`SPARSE_STATUS_ALLOC_FAILED` Internal memory allocation failed.

`SPARSE_STATUS_INVALID_VALUE` The input parameters contain an invalid value.

`SPARSE_STATUS_EXECUTION_FAILED` Execution failed.

`SPARSE_STATUS_INTERNAL_ERROR` An error in algorithm implementation occurred.

`SPARSE_STATUS_NOT_SUPPORTED` The requested operation is not supported.

mkl_sparse_?_qr_factorize

Factorization step of the SPARSE QR solver.

Syntax

```
sparse_status_t mkl_sparse_d_qr_factorize (sparse_matrix_t A, double *alt_values);
sparse_status_t mkl_sparse_s_qr_factorize (sparse_matrix_t A, float *alt_values);
```

Include Files

- `mkl_sparse_qr.h`

Description

The `mkl_sparse_?_qr_factorize` routine performs numerical factorization of matrix *A*. Prior to calling this routine, the `mkl_sparse_?_qr_reorder` routine must be called for the matrix handle *A*. For more information about the workflow of sparse QR functionality, refer to [oneMKL Sparse QR solver. Multifrontal Sparse QR Factorization Method for Solving a Sparse System of Linear Equations](#).

NOTE Currently, `mkl_sparse_?_qr_factorize` supports only CSR format for the input matrix.

Input Parameters

<i>A</i>	Handle containing a sparse matrix in an internal data structure.
<i>alt_values</i>	Array with alternative values. Must be the size of the non-zeroes in the initial input matrix. When passed to the routine, these values will be used during the factorization step instead of the values stored in handle <i>A</i> .

Return Values

`SPARSE_STATUS_SUCCESS` The operation was successful.

`SPARSE_STATUS_NOT_INITIALIZED` The routine encountered an empty handle or matrix array.

`SPARSE_STATUS_ALLOC_FAILED` Internal memory allocation failed.

`SPARSE_STATUS_INVALID_VALUE` The input parameters contain an invalid value.

`SPARSE_STATUS_EXECUTION_FAILED` Execution failed.

`SPARSE_STATUS_INTERNAL_ERROR` An error in algorithm implementation occurred.

`SPARSE_STATUS_NOT_SUPPORTED` The requested operation is not supported.

mkl_sparse?_qr_solve

Solving step of the SPARSE QR solver.

Syntax

```
sparse_status_t mkl_sparse_d_qr_solve ( sparse_operation_t operation, sparse_matrix_t
A, double *alt_values, sparse_layout_t layout, MKL_INT columns, double *x, MKL_INT ldx,
const double *b, MKL_INT ldb );

sparse_status_t mkl_sparse_s_qr_solve ( sparse_operation_t operation, sparse_matrix_t
A, float *alt_values, sparse_layout_t layout, MKL_INT columns, float *x, MKL_INT ldx,
const float *b, MKL_INT ldb );
```

Include Files

- `mkl_sparse_qr.h`

Description

The `mkl_sparse?_qr_solve` routine computes the solution of sparse systems of linear equations $A*x = b$. Prior to calling this routine, the `mkl_sparse?_qr_factorize` routine must be called for the matrix handle *A*. For more information about the workflow of sparse QR functionality, refer to [oneMKL Sparse QR solver: Multifrontal Sparse QR Factorization Method for Solving a Sparse System of Linear Equations](#).

NOTE

Currently, `mkl_sparse?_qr_solve` supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.

Alternative values are not supported and must be set to NULL.

Input Parameters

operation Specifies the operation to perform.

NOTE Currently, the only supported value is `SPARSE_OPERATION_NON_TRANSPOSE` (non-transpose case; that is, $A*x = b$ is solved).

A Handle containing a sparse matrix in an internal data structure.

alt_values

Reserved for future use.

layout

Describes the storage scheme for the dense matrix:

SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements uses column-major layout.
SPARSE_LAYOUT_ROW_MAJOR	Storage of elements uses row-major layout.

*x*Array with a size of at least *rows*cols*:

	<i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>x</i>)	<i>ldx</i>	Number of columns in <i>A</i>
<i>cols</i> (number of columns in <i>x</i>)	<i>columns</i>	<i>ldx</i>

*columns*Number of columns in matrix *b*.*ldx*Specifies the leading dimension of matrix *x*.*b*Array with a size of at least *rows*cols*:

	<i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>b</i>)	<i>ldb</i>	Number of columns in <i>A</i>
<i>cols</i> (number of columns in <i>b</i>)	<i>columns</i>	<i>ldb</i>

*ldb*Specifies the leading dimension of matrix *b*.

Output Parameters

*x*Contains the solution of system $A*x = b$.

Return Values

SPARSE_STATUS_SUCCESS

The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED

The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED

Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE

The input parameters contain an invalid value.

`SPARSE_STATUS_EXECUTION_FAILED` Execution failed.

`ON_FAILED`

`SPARSE_STATUS_INTERNAL_ERROR` An error in algorithm implementation occurred.

`L_ERROR`

`SPARSE_STATUS_NOT_SUPPORTED` The requested operation is not supported.

`PORTED`

mkl_sparse?_qr_qmult

First stage of the solving step of the SPARSE QR solver.

Syntax

```
sparse_status_t mkl_sparse_d_qr_qmult ( sparse_operation_t operation, sparse_matrix_t
A, sparse_layout_t layout, MKL_INT columns, double *x, MKL_INT ldx, const double *b,
MKL_INT ldb );
```

```
sparse_status_t mkl_sparse_s_qr_qmult ( sparse_operation_t operation, sparse_matrix_t
A, sparse_layout_t layout, MKL_INT columns, float *x, MKL_INT ldx, const float *b,
MKL_INT ldb );
```

Include Files

- `mkl_sparse_qr.h`

Description

The `mkl_sparse?_qr_qmult` routine computes multiplication of inversed matrix Q and right-hand side matrix b . This routine can be used to perform the solving step in two separate calls as an alternative to a single call of `mkl_sparse?_qr_solve`.

NOTE Currently, `mkl_sparse?_qr_qmult` supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.

Input Parameters

operation Specifies the operation to perform.

NOTE Currently, the only supported value is `SPARSE_OPERATION_NON_TRANSPOSE` (non-transpose case; that is, $A^*x = b$ is solved).

A Handle containing a sparse matrix in an internal data structure.

layout Describes the storage scheme for the dense matrix:

<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Storage of elements uses column-major layout.
<code>SPARSE_LAYOUT_ROW_MAJOR</code>	Storage of elements uses row-major layout.

x Array with a size of at least `rows*cols`:

	<i>layout =</i> SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout =</i> SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>x</i>)	<i>ldx</i>	Number of columns in <i>A</i>
<i>cols</i> (number of columns in <i>x</i>)	<i>columns</i>	<i>ldx</i>

*columns*Number of columns in matrix *b*.*ldx*Specifies the leading dimension of matrix *x*.*b*Array with a size of at least *rows*cols*:

	<i>layout =</i> SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout =</i> SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>b</i>)	<i>ldb</i>	Number of columns in <i>A</i>
<i>cols</i> (number of columns in <i>b</i>)	<i>columns</i>	<i>ldb</i>

*ldb*Specifies the leading dimension of matrix *b*.

Output Parameters

*x*Overwritten by the updated matrix $x = Q^{-1} * b$.

Return Values

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_qr_solve

Second stage of the solving step of the SPARSE QR solver.

Syntax

```
sparse_status_t mkl_sparse_d_qr_solve ( sparse_operation_t operation, sparse_matrix_t
A, sparse_layout_t layout, MKL_INT columns, double *x, MKL_INT ldx, const double *b,
MKL_INT ldb );
```

```
sparse_status_t mkl_sparse_s_qr_solve ( sparse_operation_t operation, sparse_matrix_t
A, sparse_layout_t layout, MKL_INT columns, float *x, MKL_INT ldx, const float *b,
MKL_INT ldb );
```

Include Files

- mkl_sparse_qr.h

Description

The mkl_sparse_?_qr_solve routine computes the solution of $A*x = b$.

NOTE Currently, mkl_sparse_?_qr_solve supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.

Input Parameters

operation Specifies the operation to perform.

NOTE Currently, the only supported value is SPARSE_OPERATION_NON_TRANSPOSE (non-transpose case; that is, $A*x = b$ is solved).

A Handle containing a sparse matrix in an internal data structure.

layout Describes the storage scheme for the dense matrix:

SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements uses column-major layout.
SPARSE_LAYOUT_ROW_MAJOR	Storage of elements uses row-major layout.

x Array with a size of at least $rows*cols$:

	<i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>x</i>)	<i>ldx</i>	Number of columns in <i>A</i>
<i>cols</i> (number of columns in <i>x</i>)	<i>columns</i>	<i>ldx</i>

columns Number of columns in matrix *b*.

ldx Specifies the leading dimension of matrix *x*.

b Array with a size of at least *rows*cols*:

	<i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>b</i>)	<i>ldb</i>	Number of columns in <i>A</i>
<i>cols</i> (number of columns in <i>b</i>)	<i>columns</i>	<i>ldb</i>

ldb Specifies the leading dimension of matrix *b*.

Output Parameters

x Contains the solution of the triangular system $R*x = b$.

Return Values

SPARSE_STATUS_SUCCESS The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED Execution failed.

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

Compact BLAS and LAPACK Functions

Overview

Many HPC applications rely on the application of BLAS and LAPACK operations on groups of very small matrices. While existing batch Intel® oneAPI Math Kernel Library (oneMKL) BLAS routines already provide meaningful speedup over OpenMP* loops around BLAS operations for these sizes, another customization offers potential speedup by allocating matrices in a [SIMD-friendly format](#), thus allowing for cross-matrix vectorization in the BLAS and LAPACK routines of the Intel® oneAPI Math Kernel Library (oneMKL) called *Compact BLAS and LAPACK*.

The main idea behind these compact methods is to create true SIMD computations in which subgroups of matrices are operated on with kernels that abstractly appear as scalar kernels, while registers are filled by cross-matrix vectorization.

These are the BLAS/LAPACK compact functions:

- [mkl_?gemm_compact](#)
- [mkl_?trsm_compact](#)
- [mkl_?potrf_compact](#)
- [mkl_?getrfnp_compact](#)
- [mkl_?geqrf_compact](#)
- [mkl_?getrinp_compact](#)

The compact API provides additional service functions to refactor data. Because this capability is not specific to any particular BLAS or LAPACK operation, this data manipulation can be executed once for an application's data, allowing the entire program -- consisting of any number of BLAS and LAPACK operations for which compact kernels have been written -- to be performed on the compact data without any refactoring. For applications working on data in compact format, the packing function need not be used.

See "About the Compact Format" below for more details.

Along with this new data format, the API consists of two components:

- **BLAS and LAPACK Compact Kernels:** The first component of the API is a compact kernel that works on matrices stored in compact format.
- **Service Functions for the Compact Format:** The second component of the API is a compact service function allowing for data to be factored into and out of compact format. These are:
 - [mkl_?gepack_compact](#)
 - [mkl_?geunpack_compact](#)
 - [mkl_get_format_compact](#)
 - [mkl_?get_size_compact](#)

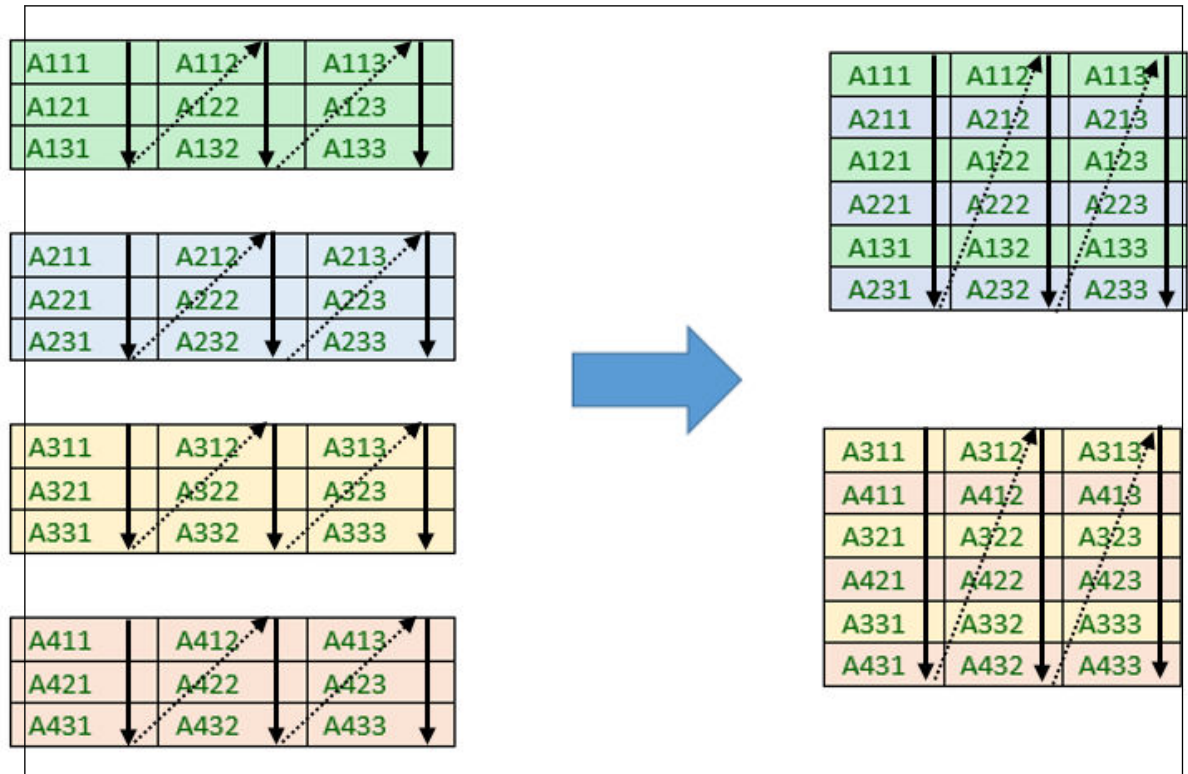
Note that there are some [Numerical Limitations](#) for the routines mentioned above.

About the Compact Format

In compact format, for calculations involving *real* precision, matrices are organized in packs of size V , where V is the SIMD vector length of the underlying architecture. Each pack is a 3D-tensor with the matrix index incrementing the fastest. These packs are then loaded into registers and operated on using SIMD instructions.

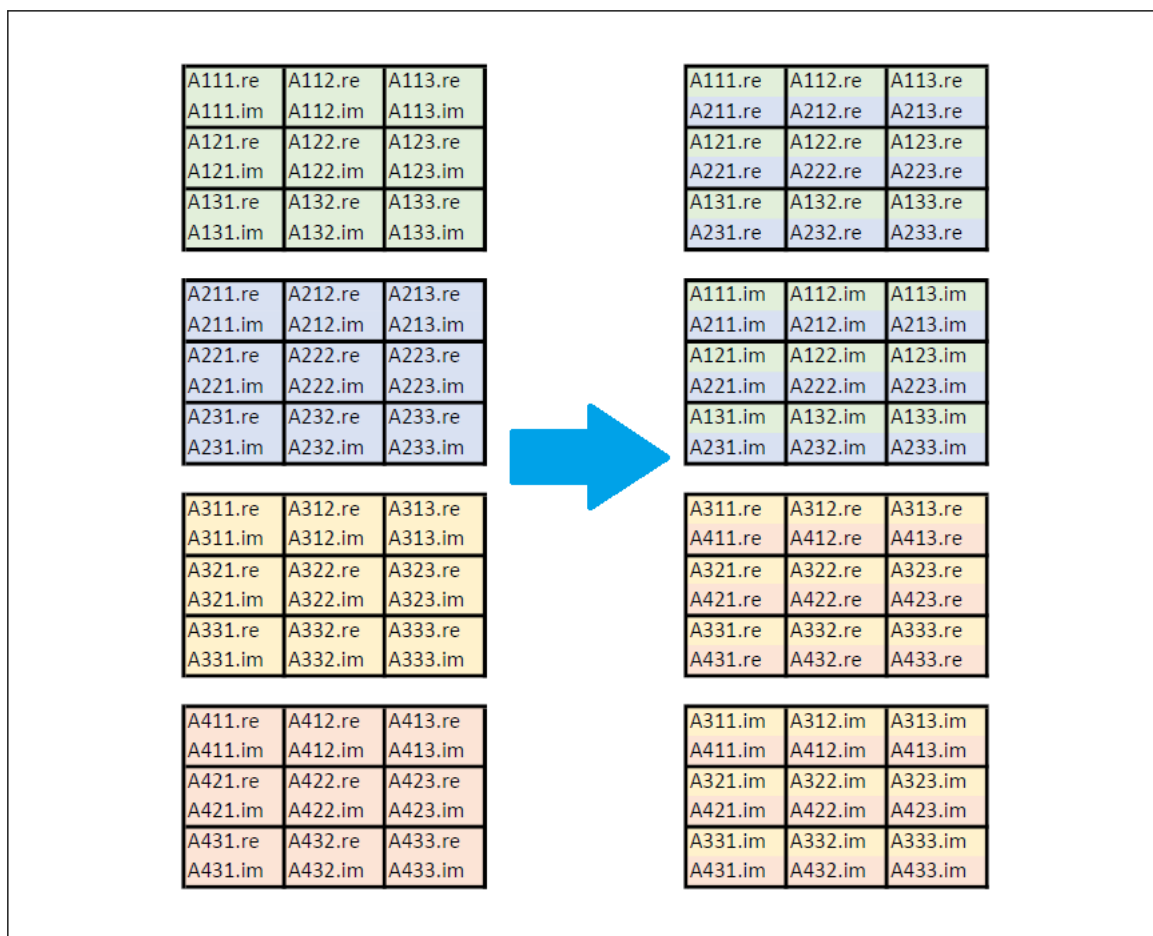
The figure below demonstrates the packing of a set of four 3 x 3 real-precision matrices into compact format. The pack length for this example is $V = 2$, resulting in 2 compact packs.

Interleaved Data for Compact BLAS and LAPACK



For calculations involving *complex* precision, the real and imaginary parts of each matrix are packed separately. In the figure below, the group of four 3 x 3 complex matrices is packed into compact format with pack length $V = 2$. The first pack consists of the real parts of the first two matrices, and the second pack consists of the imaginary parts of the first two matrices. Real and imaginary packs alternate in memory. This storage format means that *all* compact arrays can be handled as a real type.

Compact Format for Complex Precision



The particular specifications (size and number) of the compact packs for the architecture and problem-precision definition are specified by an **MKL_COMPACT_PACK** enum type. For example: given a double-precision problem involving a group of 128 matrices working on an architecture with a 256-bit SIMD vector length, the optimal pack length is $V = 4$, and the number of packs is 32.

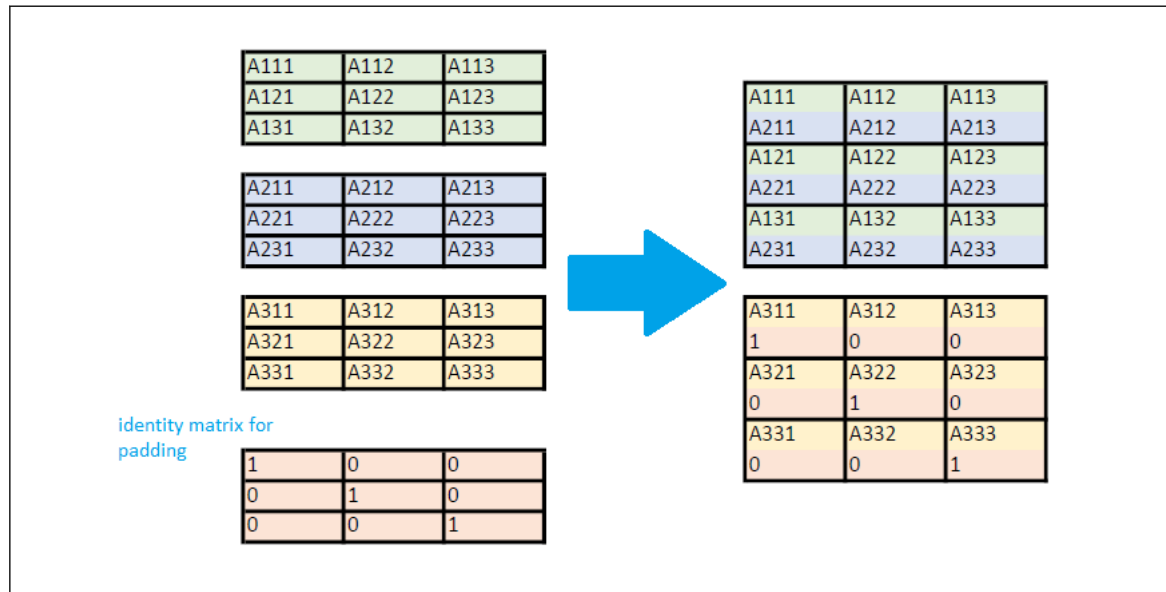
The initially-permitted values for the enum are:

- **MKL_COMPACT_SSE** - pack length 2 for double precision, pack length 4 for single precision.
- **MKL_COMPACT_AVX** - pack length 4 for double precision, pack length 8 for single precision.
- **MKL_COMPACT_AVX512** - pack length 8 for double precision, pack length 16 for single precision.

For calculations involving *complex* precision, the pack length is the same; however, half of the packs store the real parts of matrices, and half store the imaginary parts. This means that it takes **double** the number of packs to store the same number of matrices.

The above examples illustrate the case when the number of matrices is evenly-divisible by the pack length. When this is not the case, there will be partially-unfilled packs at the end of the memory segment, and the compact-packing routine will pad these partially unfilled packs with *identity matrices*, so that compact routines use only the completely-filled registers in their calculations. The next figure illustrates this padding for a group of three 3 x 3 real-precision matrices with a pack length of 2.

Compact Format with Padding



Before calling a BLAS or LAPACK compact function, the input data must be packed in compact format. After execution, the output data should be unpacked from this compact format, unless another compact routine will be called immediately following the first. Two service functions, [mkl_?gepack_compact](#), and [mkl_?geunpack_compact](#), facilitate the process of storing matrices in compact format. It is recommended that the user call the function [mkl_get_format_compact](#) before calling the [mkl_?gepack_compact](#) routine to obtain the optimal format for performance. Advanced users can pack and unpack the matrices themselves and still use Intel® oneAPI Math Kernel Library (oneMKL) compact functions on the packed set.

Compact routines can only be called for groups of matrices that have the same dimensions, leading dimension, and storage format. For example, the routine [mkl_?getrfnp_compact](#), which calculates the LU factorization of a group of $m \times n$ matrices without pivoting, can only be called for a group of matrices with the same number of rows (m) and the same number of columns (n). All of the matrices must also be stored in arrays with the same leading dimension, and all must be stored in the same storage format (column-major or row-major).

[mkl_?gemm_compact](#)

Computes a matrix-matrix product of a set of compact format general matrices.

Syntax

```
void mkl_sgemm_compact (MKL_LAYOUT layout, MKL_TRANSPOSE transa, MKL_TRANSPOSE transb,
MKL_INT m, MKL_INT n, MKL_INT k, float alpha, const float *ap, MKL_INT ldap, const float
*bp, MKL_INT ldbp, float beta, float *cp, MKL_INT ldcp, MKL_COMPACT_PACK format, MKL_INT
nm);
```

```
void mkl_dgemm_compact (MKL_LAYOUT layout, MKL_TRANSPOSE transa, MKL_TRANSPOSE transb,
MKL_INT m, MKL_INT n, MKL_INT k, double alpha, const double *ap, MKL_INT ldap, const
double *bp, MKL_INT ldbp, double beta, double *cp, MKL_INT ldcp, MKL_COMPACT_PACK
format, MKL_INT nm);
```

```
void mkl_cgemm_compact (MKL_LAYOUT layout, MKL_TRANSPOSE transa, MKL_TRANSPOSE transb,
MKL_INT m, MKL_INT n, MKL_INT k, mkl_compact_complex_float *alpha, const float *ap,
MKL_INT ldap, const float *bp, MKL_INT ldbp, mkl_compact_complex_float *beta, float
*cp, MKL_INT ldcp, MKL_COMPACT_PACK format, MKL_INT nm);
```

```
void mkl_zgemm_compact (MKL_LAYOUT layout, MKL_TRANSPOSE transa, MKL_TRANSPOSE transb,
MKL_INT m, MKL_INT n, MKL_INT k, mkl_compact_complex_double *alpha, const double *ap,
MKL_INT ldap, const double *bp, MKL_INT ldbp, mkl_compact_complex_double *beta, double
*cp, MKL_INT ldcp, MKL_COMPACT_PACK format, MKL_INT nm);
```

Description

The `mkl_?gemm_compact` routine computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product for a group of `nm` general matrices A_c that have been stored in compact format. The operation is defined for each matrix as:

$$C_c := \alpha * \text{op}(A_c) * \text{op}(B_c) + \beta * C_c$$

Where

- $\text{op}(X_c)$ is one of $\text{op}(X_c) = X_c$, or $\text{op}(X_c) = X_c^T$, or $\text{op}(X_c) = X_c^H$,
- α and β are scalars,
- A_c , B_c , and C_c are matrices that have been stored in compact format,
- $\text{op}(A_c)$ is an m -by- k matrix for each matrix in the group,
- $\text{op}(B_c)$ is a k -by- n matrix for each matrix in the group,
- and C_c is an m -by- n matrix.

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).
<i>transa</i>	Specifies the operation: If <i>transa</i> =MKL_NOTRANS, then $\text{op}(A_c) := A_c$. If <i>transa</i> =MKL_TRANS, then $\text{op}(A_c) := A_c^T$. If <i>transa</i> =MKL_CONJTRANS, then $\text{op}(A_c) := A_c^H$.
<i>transb</i>	Specifies the operation: If <i>transb</i> =MKL_NOTRANS, then $\text{op}(B_c) := B_c$. If <i>transb</i> =MKL_TRANS, then $\text{op}(B_c) := B_c^T$. If <i>transb</i> =MKL_CONJTRANS, then $\text{op}(B_c) := B_c^H$.
<i>m</i>	The number of rows of the matrices $\text{op}(A_c)$, $m \geq 0$.
<i>n</i>	The number of columns of matrices $\text{op}(B_c)$ and C_c . $n \geq 0$.
<i>k</i>	The number of columns of matrices $\text{op}(A_c)$ and the number of rows of matrices $\text{op}(B_c)$. $k \geq 0$.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .

ap

Points to the beginning of the array that stores the nmA_c matrices. See [Compact Format](#) for more details.

	<i>transa</i> =MKL_NOTTRANS	<i>transa</i> =MKL_TRANS or <i>transa</i> =MKL_CONJTRANS
<i>layout</i> = MKL_COL_MAJOR	<i>ap</i> has size $ldap*k*nm$.	<i>ap</i> has size $ldap*m*nm$.
<i>layout</i> = MKL_ROW_MAJOR	<i>ap</i> has size $ldap*m*nm$.	<i>ap</i> has size $ldap*k*nm$.

ldap

Specifies the leading dimension of A_c .

bp

Points to the beginning of the array that stores the nmB_c matrices. See [Compact Format](#) for more details.

	<i>transb</i> =MKL_NOTTRANS	<i>transb</i> =MKL_TRANS or <i>transb</i> =MKL_CONJTRANS
<i>layout</i> = MKL_COL_MAJOR	<i>bp</i> has size $ldbp*n*nm$.	<i>bp</i> has size $ldbp*k*nm$.
<i>layout</i> = MKL_ROW_MAJOR	<i>bp</i> has size $ldbp*k*nm$.	<i>bp</i> has size $ldbp*n*nm$.

ldbp

Specifies the leading dimension of B_c .

beta

Specifies the scalar *beta*.

cp

Before entry, *cp* points to the beginning of the array that stores the nmC_c matrices, except when *beta* is equal to zero, in which case *cp* need not be set on entry.

<i>layout</i> = MKL_COL_MAJOR	<i>cp</i> has size $ldap*n*nm$.
<i>layout</i> = MKL_ROW_MAJOR	<i>cp</i> has size $ldap*m*nm$.

ldcp

Specifies the leading dimension of C_c .

<i>layout</i> = MKL_COL_MAJOR	<i>ldcp</i> must be at least max (1, <i>m</i>).
<i>layout</i> = MKL_ROW_MAJOR	<i>ldcp</i> must be at least max (1, <i>n</i>).

format

Specifies the format of the compact matrices. See [Compact Format](#) or `mkl_get_format_compact` for details.

nm

Total number of matrices stored in compact format in the group of matrices.

NOTE

The values of *ldap*, *ldbp*, and *ldcp* used in `mkl_?gemm_compact` must be consistent with the values used in `mkl_?get_size_compact`, `mkl_?gepack_compact`, and `mkl_?geunpack_compact`.

Output Parameters

cp Each matrix C_c is overwritten by the m -by- n matrix ($\alpha * \text{op}(A_c) * \text{op}(B_c) + \beta * C_c$).

mkl_?trsm_compact

Solves a triangular matrix equation for a set of general, $m \times n$ matrices that have been stored in Compact format.

Syntax

```
mkl_strsm_compact (MKL_LAYOUT layout, MKL_SIDE side, MKL_UPLO uplo, MKL_TRANSPOSE
transa, MKL_DIAG diag, MKL_INT m, MKL_INT n, float alpha, const float *ap, MKL_INT
a_stride, float *bp, MKL_INT b_stride, MKL_COMPACT_PACK format, MKL_INT nm);
```

```
mkl_dtrsm_compact (MKL_LAYOUT layout, MKL_SIDE side, MKL_UPLO uplo, MKL_TRANSPOSE
transa, MKL_DIAG diag, MKL_INT m, MKL_INT n, double alpha, const double*ap, MKL_INT
a_stride, double *bp, MKL_INT b_stride, MKL_COMPACT_PACK format, MKL_INT nm);
```

```
mkl_ctrsm_compact (MKL_LAYOUT layout, MKL_SIDE side, MKL_UPLO uplo, MKL_TRANSPOSE
transa, MKL_DIAG diag, MKL_INT m, MKL_INT n, mkl_compact_complex_float *alpha, const
float *ap, MKL_INT a_stride, float *bp, MKL_INT b_stride, MKL_COMPACT_PACK format,
MKL_INT nm);
```

```
mkl_ztrsm_compact (MKL_LAYOUT layout, MKL_SIDE side, MKL_UPLO uplo, MKL_TRANSPOSE
transa, MKL_DIAG diag, MKL_INT m, MKL_INT n, mkl_compact_complex_double *alpha, const
double *ap, MKL_INT a_stride, double *bp, MKL_INT b_stride, MKL_COMPACT_PACK format,
MKL_INT nm);
```

Description

The routine solves one of the following matrix equations for a group of *nm* matrices:

$$\text{op}(A_c) * X_c = \alpha * B_c,$$

or

$$X_c * \text{op}(A_c) = \alpha * B_c$$

where:

α is a scalar, X_c and B_c are m -by- n matrices that have been stored in compact format, and A_c is a m -by- m unit, or non-unit, upper or lower triangular matrix that has been stored in compact format.

$\text{op}(A_c)$ is one of $\text{op}(A_c) = A_c$, or $\text{op}(A_c) = A_c^T$, or $\text{op}(A_c) = A_c^H$,

B_c is overwritten by the solution matrix X_c .

Input Parameters

layout Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).

side Specifies whether $\text{op}(A_c)$ appears on the left or right of X_c in the equation:

if side = MKL_LEFT, then $\text{op}(A_c) * X_c = \alpha * B_c$, if side = MKL_RIGHT, then $X_c * \text{op}(A_c) = \alpha * B_c$

uplo

Specifies whether matrix A_c is upper or lower triangular.

If uplo = MKL_UPPER, A_c is upper triangular.

If uplo = MKL_LOWER, A_c is lower triangular.

transa

Specifies the operation:

If *transa*=MKL_NOTRANS, then $\text{op}(A_c) = A_c$;

If *transa*=MKL_TRANS, then $\text{op}(A_c) = A_c^T$;

If *transa*=MKL_CONJTRANS, then $\text{op}(A_c) = A_c^H$;

diag

Specifies whether the matrix A_c is unit triangular:

If *diag*=MKL_UNIT, then the matrix is unit triangular;

if *diag*=MKL_NONUNIT, then the matrix is not unit triangular.

m

The number of rows of B_c and the number of rows and columns of A_c when *side*=MKL_LEFT; $m \geq 0$.

n

The number of columns of B_c and the number of rows and columns of A_c when *side*=MKL_RIGHT; $n \geq 0$.

alpha

Specifies the scalar *alpha*. When alpha is zero, then ap is not referenced and bp need not be set before entry.

ap

Array, size $ldap * k * nm$, where k is m when *side*= MKL_LEFT and n when *side* = MKL_RIGHT. *ap* points to the beginning of nm A_c matrices stored in compact format. When uplo = MKL_UPPER, A_c is assumed to be an upper triangular matrix and the lower triangular part of A_c is not referenced. With uplo = MKL_LOWER, A_c is assumed to be a lower triangular matrix and the upper triangular part of A_c is not referenced. With *diag* = MKL_UNIT, the diagonal elements of A_c are not referenced either, but are assumed to be unity.

ldap

Column stride (column-major layout) or row stride (row-major layout) of A_c .

When *side*=MKL_LEFT, *ldap* must be at least $\max(1, m)$.

When *side*=MKL_RIGHT, *ldap* must be at least $\max(1, n)$.

bp

Array, size $ldbp * n * nm$ when layout = MKL_COL_MAJOR; size $ldbp * m * nm$ when layout = MKL_ROW_MAJOR. Before entry, bp points to the beginning of nm B_c matrices stored in compact format.

ldbp

Column stride (column-major layout) or row stride (row-major layout) of B_c .

<i>layout</i> = MKL_COL_MAJOR	<i>ldbp</i> must be at least $\max(1, m)$.
<i>layout</i> = MKL_ROW_MAJOR	* <i>ldbp</i> must be at least $\max(1, n)$.

format

Specifies the format of the compact matrices. See <Compact Format> or `mkl_get_format_compact` for details.

nm

Total number of matrices stored in Compact format; $nm \geq 0$.

NOTE

The values of `ldap` and `ldbp` used in `mkl_?trsm_compact` must be consistent with the values used in `mkl_?get_size_compact`, `mkl_?gepack_compact`, and `mkl_?geunpack_compact`.

Output Parameters

`bp` On exit, B_c is overwritten by the solution matrix X_c . `bp` points to the beginning of `nm` such X_c matrices.

mkl_?potrf_compact

Computes the Cholesky factorization of a set of symmetric (Hermitian), positive-definite matrices, stored in Compact format (see [Compact Format](#) for details).

Syntax

```
void mkl_spotrf_compact (MKL_LAYOUT layout, MKL_UPLO uplo, MKL_INT n, float * ap,
MKL_INT ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_cpotrf_compact (MKL_LAYOUT layout, MKL_UPLO uplo, MKL_INT n, float * ap,
MKL_INT ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_dpotrf_compact (MKL_LAYOUT layout, MKL_UPLO uplo, MKL_INT n, double * ap,
MKL_INT ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_zpotrf_compact (MKL_LAYOUT layout, MKL_UPLO uplo, MKL_INT n, double * ap,
MKL_INT ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
```

Description

The routine forms the Cholesky factorization of a set of symmetric, positive definite (or, for complex data, Hermitian, positive-definite), $n \times n$ matrices A_c , stored in Compact format, as:

- $A_c = U_c^T U_c$ (for real data), $A_c = U_c^H U_c$ (for complex data), if `uplo = MKL_UPPER`
- $A_c = L_c L_c^T$ (for real data), $A_c = L_c L_c^H$ (for complex data), if `uplo = MKL_LOWER`

where L_c is a lower triangular matrix, and U_c is upper triangular. The factorization (output) data will also be stored in Compact format.

Before calling this routine, call `mkl_?gepack_compact` to store the matrices in the Compact format.

NOTE

Compact routines have some limitations; see [Numerical Limitations](#).

Input Parameters

<code>layout</code>	Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).
<code>uplo</code>	<p>Must be MKL_UPPER or MKL_LOWER</p> <p>Indicates whether the upper or lower triangular part of A_c has been stored and will be factored.</p> <p>If <code>uplo = MKL_UPPER</code>, the upper triangular part of A_c is stored, and the strictly lower triangular part of A_c is not referenced.</p>

	If <code>uplo = MKL_LOWER</code> , the lower triangular part of A_c is stored, and the strictly upper triangular part of A_c is not referenced.
<code>n</code>	The order of A_c ; $n \geq 0$.
<code>ldap</code>	Column stride (column-major layout) or row stride (row-major layout) of A_c .
<code>ap</code>	Points to the beginning of the nm A_c matrices. On entry, <code>ap</code> contains either the upper or the lower triangular part of A_c (see <code>uplo</code>).
<code>format</code>	Specifies the format of the compact matrices. See Compact Format or mkl_get_format_compact for details.
<code>nm</code>	Total number of matrices stored in Compact format; $nm \geq 0$.

Application Notes:

Before calling this routine, `mkl_?gepack_compact` must be called. After calling this routine, `mkl_?geunpack_compact` should be called, unless another compact routine will be called for the Compact format matrices.

The total number of floating-point operations is approximately $nm * (1/3) n^3$ for real flavors and $nm * (4/3) n^3$ for complex flavors.

Output Parameters

<code>ap</code>	The upper or lower triangular part of A_c , stored in Compact format in <code>ap</code> , is overwritten by its Cholesky factor U_c or L_c (as specified by <code>uplo</code>). <code>ap</code> now points to the beginning of this set of factors, stored in Compact format.
<code>info</code>	The parameter is not currently used in this routine. It is reserved for the future use.

`mkl_?getrfnp_compact`

The routine computes the LU factorization, without pivoting, of a set of general, $m \times n$ matrices that have been stored in Compact format (see [Compact Format](#)).

Syntax

```
void mkl_sgetrfnp_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, float * ap, MKL_INT
ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_dgetrfnp_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, double * ap,
MKL_INT ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_cgetrfnp_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, float * ap, MKL_INT
ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_zgetrfnp_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, double * ap,
MKL_INT ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
```

Description

The `mkl_?getrfnp_compact` routine calculates the LU factorizations of a set of `nm` general ($m \times n$) matrices A , stored in Compact format, as $A_c = L_c * U_c$. The factorization (output) data will also be stored in Compact format.

NOTE

Compact routines have some limitations; see [Numerical Limitations](#).

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).
<i>m</i>	The number of rows of A; $m \geq 0$.
<i>n</i>	The number of columns of A; $n \geq 0$.
<i>ap</i>	Points to the beginning of the the array which stores nm A_c matrices. See Compact Format for more details.
<i>ldap</i>	Leading dimension of A_c .
<i>format</i>	Specifies the format of the compact matrices. See Compact Format or <code>mkl_get_format_compact</code> for details.
<i>nm</i>	Total number of matrices stored in Compact format.

Application Notes:

Before calling this routine, [mkl_?gepack_compact](#) must be called. After calling this routine, [mkl_?geunpack_compact](#) should be called, unless another compact routine will be subsequently called for the Compact format matrices.

The approximate number of floating-point operations for real flavors is:

$nm \cdot (2/3)n^3$, if $m = n$,

$nm \cdot (1/3)n^2(3m-n)$, if $m > n$,

$nm \cdot (1/3)m^2(3n-m)$, if $m < n$.

The number of operations for complex flavors is four times greater. Directly after calling this routine, you can call the following:

[mkl_?getrinp_compact](#), for computing the inverse of the *nm* input matrices in Compact format

Output Parameters

<i>ap</i>	On exit, A_c is overwritten by its factorization data. <i>ap</i> points to the beginning of <i>nm</i> L_c and U_c factors of A_c . The unit diagonal elements of L_c are not stored.
<i>info</i>	The parameter is not currently used in this routine. It is reserved for the future use.

[mkl_?geqrf_compact](#)

Computes the QR factorization of a set of general $m \times n$, matrices, stored in Compact format (see [Compact Format](#) for details).

Syntax

```
void mkl_sgeqrf_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, float * ap, MKL_INT ldap, float * tau, float * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
```

```

void mkl_cgeqrf_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, float * ap, MKL_INT
ldap, float * taup, float * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK
format, MKL_INT nm);

void mkl_dgeqrf_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, double * ap, MKL_INT
ldap, double * taup, double * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK
format, MKL_INT nm);

void mkl_zgeqrf_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, double * ap, MKL_INT
ldap, double * taup, double * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK
format, MKL_INT nm);

```

Description

The routine forms the QR factorization of a set of general, $m \times n$ matrices A , stored in Compact format. The routine does not form the Q factors explicitly. Instead, Q is represented as a product of $\min(m,n)$ elementary reflectors. The factorization (output) data will also be stored in Compact format.

NOTE

Compact routines have some limitations; see [Numerical Limitations](#).

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).
<i>m</i>	The number of rows of A_c ; $m \geq 0$.
<i>n</i>	The number of columns of A_c ; $n \geq 0$.
<i>ap</i>	Points to the beginning of the nm A_c matrices. On entry, <i>ap</i> contains either the upper or the lower triangular part of A_c (see <i>uplo</i>).
<i>ldap</i>	Column stride (column-major layout) or row stride (row-major layout) of A_c .
<i>work</i>	Points to the beginning of the workspace array.
<i>lwork</i>	The size of the work array. If <i>lwork</i> = -1, a workspace query is assumed; the routine only calculates the optimal size of the work array and returns this value as the first entry of the work array.
<i>format</i>	Specifies the format of the compact matrices. See Compact Format or mkl_get_format_compact for details.
<i>nm</i>	Total number of matrices stored in Compact format.

Application Notes:

The compact array that will store the elementary reflectors needs to be allocated before the routine is called and unpacked after. First, the routine [mkl_get_size_compact](#) should be called, to determine the size of *taup*, and memory for *taup* should be allocated. After calling [mkl_geqrf_compact](#), *taup* stores the elementary reflectors in compact form, so should be unpacked using [mkl_geunpack_compact](#). See [Compact Format](#) for more details, or reference the example below. (Note: the following example is meant to demonstrate the calling sequence to allocate memory and unpack *taup*. All other parameters are assumed to be already set up before the sequence below is executed.)

```

MKL_R_TYPE *tau_array[nm];
// ...
tau_buffer_size = mkl_get_size_compact(min(m, n), 1, format, nm);

```

```

MKL_R_TYPE *tau_compact = (MKL_R_TYPE *)mkl_malloc(tau_buffer_size, 128);
mkl_?geqrf_compact(layout, m, n, a_compact, ldap, tau_compact, work, lwork, &info, format, nm);
// Note that here MKL_COL_MAJOR is used because tau is a 1-d array
mkl_?geunpack_compact(MKL_COL_MAJOR, min(m, n), 1, tau_array, min(m, n), tau_compact, min(m, n),
format, nm);

```

Output Parameters

<i>ap</i>	On exit, A_c is overwritten by its factorization data. <i>ap</i> points to the beginning of $n \times m$ factorizations of A_c , stored in Compact format. The factorization data is stored as follows: The elements on and above the diagonal contain the $\min(m, n)$ -by- n upper trapezoidal matrix R_c (R_c is upper triangular if $m \geq n$); the elements below the diagonal, with <i>tau</i> , present the orthogonal matrix Q_c as a product of $\min(m, n)$ elementary reflectors (see Orthogonal Factorizations: LAPACK Computational Routines). See Compact Format for more details.
<i>taup</i>	Points to the beginning of a set of the τ_c arrays, each of which has size $\min(m, n)$, stored in Compact format. τ_c contains scalars that define elementary reflectors for Q_c in its decomposition in a product of elementary reflectors. <i>taup</i> needs to be allocated by the user before calling this routine. See the application notes (below the description) for more details.
<i>work[0]</i>	On exit contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	The parameter is not currently used in this routine. It is reserved for the future use.

mkl_?getrinp_compact

Computes the inverse of a set of LU-factorized general matrices, without pivoting, stored in the compact format (see [Compact Format](#) for details).

Syntax

```

void mkl_sgetrinp_compact (MKL_LAYOUT layout, MKL_INT n, float * ap, MKL_INT ldap,
float * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
void mkl_dgetrinp_compact (MKL_LAYOUT layout, MKL_INT n, double * ap, MKL_INT ldap,
double * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
void mkl_cgetrinp_compact (MKL_LAYOUT layout, MKL_INT n, float * ap, MKL_INT ldap,
float * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
void mkl_zgetrinp_compact (MKL_LAYOUT layout, MKL_INT n, double * ap, MKL_INT ldap,
double * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);

```

Description

This routine computes the inverse $\text{inv}(A_c)$ of a set of general, $n \times n$ matrices A_c , that have been stored in Compact format. The factorization (output) data will also be stored in Compact format.

NOTE

Compact routines have some limitations; see [Numerical Limitations](#).

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).
<i>n</i>	The order of A_c ; $n \geq 0$.
<i>ap</i>	Points to the beginning of the nm A_c matrices. On entry, <i>ap</i> contains the LU factorizations of A_c , stored in Compact format, as returned by mkl_?getrfnp_compact : $A_c = L_c * U_c$. See Compact Format for more details.
<i>ldap</i>	Column stride (column-major layout) or row stride (row-major layout) of A_c .
<i>work</i>	Points to the beginning of the <i>work</i> array.
<i>lwork</i>	The size of the work array. If <i>lwork</i> = -1, a workspace query is assumed; the routine calculates only the optimal size of the work array and returns this value as the first entry of the work array.
<i>format</i>	Specifies the format of the compact matrices. See Compact Format or mkl_get_format_compact for details.
<i>nm</i>	Total number of matrices stored in Compact format.

Application Notes:

Before calling this routine, [mkl_?gepack_compact](#) must be called. After calling this routine, [mkl_?geunpack_compact](#) should be called, unless another compact routine will be subsequently called on the Compact format matrices.

The total number of floating-point operations is approximately $nm * (4/3) n^3$ for real flavors and $nm * (16/3) n^3$ for complex flavors.

Output Parameters

<i>ap</i>	On exit, A_c is overwritten by $\text{inv}(A_c)$. <i>ap</i> points to the beginning of nm $\text{inv}(A_c)$ matrices stored in Compact format.
<i>work[0]</i>	On exit, <i>work[0]</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	The parameter is not currently used in this routine. It is reserved for the future use.

Numerical Limitations for Compact BLAS and Compact LAPACK Routines

Compact routines are subject to a set of numerical limitations. They also skip most of the checks presented in regular BLAS and LAPACK routines in order to provide effective vectorization. The following limitations apply to at least one compact routine.

Complex division: BLAS and LAPACK compact routines rely on a naïve method for complex division that does not protect the solution against overflow, underflow, or loss of precision.

Error checking : the LAPACK compact routines skip error checking for performance reasons ; therefore, the user is responsible for passing correct parameters. There are no checks for incorrect matrices (such as singular for LU, non-positive-definite for Cholesky) - it is always assumed that the algorithm for the input matrix can be completed without error.

No pivoting: the generic LU factorization routine, [?getrf](#) , calculates the factorization using partial pivoting. However, because pivoting includes comparisons which cannot be effectively vectorized, only non-pivoting versions of LU [mkl_?getrfnp_compact](#) and Inverse from LU ([mkl_?getrinp_compact](#)) are provided as compact routines.

Matrices scaled near underflow/overflow: the LAPACK compact routines do not provide safe handling for values near underflow/overflow. This means that Compact routines may return incorrect results for such matrices. This limitation is related to compact routine for QR: `mkl_?geqrf_compact`.

It is the responsibility of the user to ensure that the input matrices can be factorized, inverted, and/or solved given these numerical limitations.

`mkl_?get_size_compact`

Returns the buffer size, in bytes, needed to pack data in Compact format.

Syntax

```
MKL_INT mkl_sget_size_compact (MKL_INT ld, MKL_INT sd, MKL_COMPACT_PACK format, MKL_INT nm);
```

```
MKL_INT mkl_dget_size_compact (MKL_INT ld, MKL_INT sd, MKL_COMPACT_PACK format, MKL_INT nm);
```

```
MKL_INT mkl_cget_size_compact (MKL_INT ld, MKL_INT sd, MKL_COMPACT_PACK format, MKL_INT nm);
```

```
MKL_INT mkl_zget_size_compact (MKL_INT ld, MKL_INT sd, MKL_COMPACT_PACK format, MKL_INT nm);
```

Description

The routine returns the buffer size, in bytes, required for `mkl_?gepack_compact`.

Input Parameters

<i>ld</i>	Leading dimension of the matrices in Compact format.
<i>sd</i>	Second dimension of the matrices in Compact format.
<i>format</i>	Describes the compact packing format according to the MKL_COMPACT_PACK enum type.
<i>nm</i>	Total number of matrices to be packed in Compact format.

Application Notes:

Before calling this routine, `mkl_?get_format_compact` can be called to determine the optimal *format*.

After calling this routine and allocating the amount of memory indicated by *size*, the user can call `mkl_?gepack_compact` to pack the *nm* input matrices in Compact format.

Return Values

This function returns a value *size*.

<i>size</i>	The buffer size, in bytes, required by the packing function <code>mkl_?gepack_compact</code> .
-------------	--

`mkl_get_format_compact`

Returns the optimal compact packing format for the architecture, needed for all compact routines.

Syntax

```
MKL_COMPACT_PACK mkl_get_format_compact ();
```

Description

The routine returns the optimal compact packing *format*, which is an `MKL_COMPACT_PACK` type, for the current architecture. The optimal value of *format* is determined by the architecture's vector-register length. *format* is a required parameter for any packing, unpacking, or BLAS/LAPACK compact routine. See [Compact Format](#) for details.

Return Values

The function returns a value *format*.

format

format can be returned as any of the following three values. **MKL_COMPACT_AVX512** is the optimal *format* value for:

- Intel® Advanced Vector Extensions 512 (Intel® AVX-512)-enabled processors.
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) for Intel® Many Integrated Core Architecture (Intel® MIC Architecture)-enabled processors.
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) for Intel® Many Integrated Core Architecture (Intel® MIC Architecture) with support of AVX512_4FMAPS and AVX512_4VNNIW instruction groups processors.

MKL_COMPACT_AVX is the optimal *format* value for:

- Intel® Advanced Vector Extensions (Intel® AVX)-enabled processors.
- Intel® Advanced Vector Extensions 2 (Intel® AVX2)-enabled processors.

MKL_COMPACT_SSE is the optimal *format* value for all other processors.

Application Notes:

After calling this routine, `mkl_?get_size_compact` can be called to calculate the buffer size needed for `mkl_?gepack_compact`.

`mkl_?gepack_compact`

Packs matrices from standard (row or column-major) format to Compact format.

Syntax

```
mkl_sgepack_compact(MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, const float *
const *a, MKL_INT lda, float *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);

mkl_dgepack_compact(MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, const double *
const *a, MKL_INT lda, double *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);

mkl_cgepack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, const
mkl_compact_complex_float * const *a, MKL_INT lda, float *ap, MKL_INT ldap,
MKL_COMPACT_PACK format, const MKL_INT nm);
```

```
mkl_zgepack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, const
mkl_compact_complex_double * const *a, MKL_INT lda, double *ap, MKL_INT ldap,
MKL_COMPACT_PACK format, MKL_INT nm);
```

Description

The routine packs nm matrices A from standard format (row or column-major, pointer to pointer) in a into Compact format, storing the new compact format matrices A_c in array ap .

Input Parameters

layout Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).

rows The number of rows of A ; $rows \geq 0$.

columns The number of columns of A ; $columns \geq 0$.

a A standard format (row or column-major, pointer-to-pointer) array, storing nm input A matrices.

lda Leading dimension of A .

<i>layout</i> = MKL_COL_MAJOR	<i>lda</i> must be at least $\max(1, rows)$.
<i>layout</i> = MKL_ROW_MAJOR	<i>lda</i> must be at least $\max(1, columns)$.

ldap Leading dimension of A_c .

<i>layout</i> = MKL_COL_MAJOR	<i>ldap</i> must be at least $\max(1, rows)$.
<i>layout</i> = MKL_ROW_MAJOR	<i>ldap</i> must be at least $\max(1, columns)$.

NOTE

The values of *ldap* used in `mkl_?gepack_compact` must be consistent with the values used in `mkl_?get_size_compact` and `mkl_?geunpack_compact`.

format Specifies the format of the compact matrices. See [Compact Format](#) or `mkl_get_format_compact` for details.

nm Total number of matrices that will be stored in Compact format.

Application Notes:

Directly after calling this routine, any BLAS or LAPACK compact routine can be called. Unpacking matrices from Compact format can be done by calling `mkl_?geunpack_compact`.

Output Parameters

ap Array storing the compact format input matrices A_c . *ap* must have size at least $size = mkl_?get_size_compact$.

mkl_?geunpack_compact

Unpacks matrices from Compact format to standard (row- or column-major, pointer-to-pointer) format.

Syntax

```
mkl_sgeunpack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, float * const
*a, MKL_INT lda, const float *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);

mkl_dgeunpack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, double * const
*a, MKL_INT lda, const double *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);

mkl_cgeunpack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns,
mkl_compact_complex_float * const *a, MKL_INT lda, const float *ap, MKL_INT ldap,
MKL_COMPACT_PACK format, MKL_INT nm);

mkl_zgeunpack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns,
mkl_compact_complex_double * const *a, MKL_INT lda, const double *ap, MKL_INT ldap,
MKL_COMPACT_PACK format, MKL_INT nm);
```

Description

The routine unpacks nm Compact format matrices A_c from array ap into standard (row- or column-major, pointer-to-pointer) format in array A .

Input Parameters

layout Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).

rows The number of rows of A ; $rows \geq 0$.

columns The number of columns of A ; $columns \geq 0$.

lda Leading dimension of A .

$layout = \text{MKL_COL_MAJOR}$	lda must be at least $\max(1, rows)$.
$layout = \text{MKL_ROW_MAJOR}$	lda must be at least $\max(1, columns)$.

ap Array storing the compact format of input matrices A_c . See [Compact Formator](#) [mkl_get_format_compact](#) for details.

$layout = \text{MKL_COL_MAJOR}$	ap has size $ldap * columns * nm$.
$layout = \text{MKL_ROW_MAJOR}$	ap has size $ldap * rows * nm$.

ldap Leading dimension of A_c .

$layout = \text{MKL_COL_MAJOR}$	$ldap$ must be at least $\max(1, rows)$.
$layout = \text{MKL_ROW_MAJOR}$	$ldap$ must be at least $\max(1, columns)$.

NOTE
The values of *ldap* used in `mkl_?geunpack_compact` must be consistent with the values used in `mkl_?get_size_compact` and `mkl_?gepack_compact`.

format Specifies the format of the compact matrices. See [Compact Format](#) or `mkl_get_format_compact` for details.

nm Total number of matrices that will be stored in Compact format.

Output Parameters

a A standard format (row- or column-major, pointer-to-pointer) array, storing *nm* output *A* matrices.

<i>layout</i> = MKL_COL_MAJOR	<i>a</i> has size <i>lda</i> * <i>columns</i> * <i>nm</i> .
<i>layout</i> = MKL_ROW_MAJOR	<i>a</i> has size <i>lda</i> * <i>rows</i> * <i>nm</i> .

Inspector-executor Sparse BLAS Routines

The inspector-executor API for Sparse BLAS divides operations into two stages: analysis and execution. During the initial analysis stage, the API inspects the matrix sparsity pattern and applies matrix structure changes. In the execution stage, subsequent routine calls reuse this information in order to improve performance.

The inspector-executor API supports key Sparse BLAS operations for iterative sparse solvers:

- Sparse matrix-vector multiplication
- Sparse matrix-matrix multiplication with a sparse or dense result
- Solution of triangular systems
- Sparse matrix addition

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

Naming Conventions in Inspector-Executor Sparse BLAS Routines

The Inspector-Executor Sparse BLAS API routine names use the following convention:

`mkl_sparse_[<character>_]<operation>[_<format>]`

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

- | | |
|---|---------------------------|
| s | real, single precision |
| c | complex, single precision |
| d | real, double precision |
| z | complex, double precision |

The data type is included in the name only if the function accepts dense matrix or scalar floating point parameters.

The *<operation>* field indicates the type of operation:

<code>create</code>	create matrix handle
<code>copy</code>	create a copy of matrix handle
<code>convert</code>	convert matrix between sparse formats
<code>export</code>	export matrix from internal representation to CSR or BSR format
<code>destroy</code>	frees memory allocated for matrix handle
<code>set_<op>_hint</code>	provide information about number of upcoming compute operations and operation type for optimization purposes, where <i><op></i> is <code>mv</code> , <code>sv</code> , <code>mm</code> , <code>sm</code> , <code>dotmv</code> , <code>symgs</code> , or <code>memory</code>
<code>optimize</code>	analyze the matrix using hints and store optimization information in matrix handle
<code>mv</code>	compute sparse matrix-vector product
<code>mm</code>	compute sparse matrix by dense matrix product (batch <code>mv</code>)
<code>set_value</code>	change a value in a matrix
<code>sppmm/sppmmd</code>	compute sparse matrix by sparse matrix product and store the result as a sparse/dense matrix
<code>trsv</code>	solve a triangular system
<code>trsm</code>	solve a triangular system with multiple right-hand sides
<code>add</code>	compute sum of two sparse matrices
<code>symgs</code>	compute a symmetric Gauss-Zeidel preconditioner
<code>symgs_mv</code>	compute a symmetric Gauss-Zeidel preconditioner with a final matrix-vector multiplication
<code>sorv</code>	computes forward, backward sweeps or symmetric successive over-relaxation preconditioner
<code>sypr</code>	compute the symmetric or Hermitian product of sparse matrices and store the result as a sparse matrix
<code>syprd</code>	compute the symmetric or Hermitian product of sparse and dense matrices and store the result as a dense matrix
<code>syrk</code>	compute the product of sparse matrix with its transposed matrix and store the result as a sparse matrix
<code>syrkd</code>	compute the product of sparse matrix with its transposed matrix and store the result as a dense matrix
<code>order</code>	perform ordering of column indexes of the matrix in CSR format
<code>dotmv</code>	compute a sparse matrix-vector product with dot product

The *<format>* field indicates the sparse matrix storage format:

<code>coo</code>	coordinate format
<code>bsr</code>	block sparse row format plus variations. Fill out either <code>rows_start</code> and <code>rows_end</code> (for 4-arrays representation) or <code>rowIndex</code> array (for 3-array BSR/CSR).

<code>csr</code>	compressed sparse row format plus variations. Fill out either <code>rows_start</code> and <code>rows_end</code> (for 4-arrays representation) or <code>rowIndex</code> array (for 3-array BSR/CSR).
<code>csc</code>	compressed sparse column format plus variations. Fill out either <code>cols_start</code> and <code>cols_end</code> (for 4-arrays representation) or <code>colIndex</code> array (for 3 array CSC).

The format is included in the function name only if the function parameters include an explicit sparse matrix in one of the conventional sparse matrix formats.

Sparse Matrix Storage Formats for Inspector-executor Sparse BLAS Routines

Inspector-executor Sparse BLAS routines support four conventional sparse matrix storage formats:

- compressed sparse row format (CSR) plus variations
- compressed sparse column format (CSC) plus variations
- coordinate format (COO)
- block sparse row format (BSR) plus variations

Computational routines operate on a matrix handle that stores a matrix in CSR or BSR formats. Other formats should be converted to CSR or BSR format before calling any computational routines. For more information see [Sparse Matrix Storage Formats](#).

Supported Inspector-executor Sparse BLAS Operations

The Inspector-executor Sparse BLAS API can perform several operations involving sparse matrices. These notations are used in the description of the operations:

- A , G , V are sparse matrices
- B and C are dense matrices
- x and y are dense vectors
- α and β are scalars

$\text{op}(A)$ represents a possible transposition of matrix A

$$\begin{aligned}\text{op}(A) &= A \\ \text{op}(A) &= A^T - \text{transpose of } A \\ \text{op}(A) &= A^H - \text{conjugate transpose of } A\end{aligned}$$

$\text{op}(A)^{-1}$ denotes the inverse of $\text{op}(A)$.

The Inspector-executor Sparse BLAS routines support the following operations:

- computing the vector product between a sparse matrix and a dense vector:

```
y := alpha*op(A)*x + beta*y
```

- solving a single triangular system:

```
y := alpha*inv(op(A))*x
```

- computing a product between a sparse matrix and a dense matrix:

```
C := alpha*op(A)*B + beta*C
```

- computing a product between sparse matrices with a sparse result:

```
V := alpha*op(A)*op(G)
```

- computing a product between sparse matrices with a dense result:

```
C := alpha*op(A)*op(G)
```

- computing a sum of sparse matrices with a sparse result:

```
V := alpha*op(A) + G
```

- solving a sparse triangular system with multiple right-hand sides:

```
C := alpha*inv(op(A))*B
```

Two-stage Algorithm in Inspector-Executor Sparse BLAS Routines

You can use a two-stage algorithm in Inspector-executor Sparse BLAS routines which produce a sparse matrix. The applicable routines are:

- `mkl_sparse_sp2m` (BSR/CSR/CSC formats)
- `mkl_sparse_sypr` (CSR format)

The two-stage algorithm allows you to split computations into stages. The main purpose of the splitting is to provide an estimate for the memory required for the output prior to allocating the largest part of the memory (for the indices and values of the non-zero elements). Additionally, the two-stage approach extends the functionality and allows more complex usage models.

NOTE The multistage approach currently does not allow you to allocate memory for the output matrix outside oneMKL.

In the two-stage algorithm:

1. The first stage allocates data which is necessary for the memory estimation (arrays `rows_start/rows_end` or `cols_start/cols_end` depending on the format, (see [Sparse Matrix Storage Formats](#)) and computes the number of entries or the full structure of the matrix.

NOTE The format of the output is decided internally but can be checked using the export functionality `mkl_sparse_?_export_<format>`.

2. The second stage allocates data and computes column or row indices (depending on the format) of non-zero elements and/or values of the output matrix.

Specifying the stage for execution is supported through the `sparse_request_t` parameter in the API with the following options:

Values for `sparse_request_t` parameter

Value	Description
<code>SPARSE_STAGE_NNZ_COUNT</code>	Allocates and computes only the <code>rows_start/rows_end</code> (CSR/BSR format) or <code>cols_start/cols_end</code> (CSC format) arrays for the output matrix. After this stage, by calling <code>mkl_sparse_?_export_<format></code> , you can obtain the number of non-zeros in the output matrix and calculate the amount of memory required for the output matrix.
<code>SPARSE_STAGE_FINALIZE_MULT_NO_VAL</code>	Allocates and computes row/column indices provided that <code>rows_start/rows_end</code> or <code>cols_start/cols_end</code> have already been computed in a prior call with the request <code>SPARSE_STAGE_NNZ_COUNT</code> . The values of the output matrix are not computed.
<code>SPARSE_STAGE_FINALIZE_MULT</code>	Depending on the state of the output matrix C on entry to the routine, this stage does one of the following: <ul style="list-style-type: none"> • Allocates and computes row/column indices and values of nonzero elements, if only <code>rows_start/rows_end</code> or <code>cols_start/cols_end</code> are present • allocates and computes values of nonzero elements, if <code>rows_start/rows_end</code> or <code>cols_start/cols_end</code> and row/column indices of non-zero elements are present
<code>SPARSE_STAGE_FULL_MULT_NO_VAL</code>	Allocates and computes the output matrix structure in a single step. The values of the output matrix are not computed.
<code>SPARSE_STAGE_FULL_MULT</code>	Allocates and computes the entire output matrix (structure and values) in a single step.

The example below shows how you can use the two-stage approach for estimating the memory requirements for the output matrix in CSR format:

First stage (`sparse_request_t = SPARSE_STAGE_NNZ_COUNT`)

1. The routine `mkl_sparse_sp2m` is called with the request parameter `SPARSE_STAGE_NNZ_COUNT`.
2. The arrays `rows_start` and `rows_end` are exported using the `mkl_sparse_x_export_csr` routine.
3. These arrays are used to calculate the number of non-zeros (nnz) of the resulting output matrix.

Note that by the end of the first stage, the arrays associated with column indices and values of the output matrix have not been allocated or computed yet.

```
sparse_matrix_t csrC = NULL;
status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB, csrB, SPARSE_STAGE_NNZ_COUNT, &csrC);

/* optional calculation of nnz in the output matrix for getting a memory estimate */

status = mkl_sparse_?_export_csr (csrC, &indexing, &nrows, &ncols, &rows_start, &rows_end,
&col_indx, &values);

MKL_INT nnz = rows_end[nrows-1] - rows_start[0];
```

Second stage (`sparse_request_t = SPARSE_STAGE_FINALIZE_MULT`)

This stage allocates and computes the remaining output arrays (associated with column indices and values of output matrix entries) and completes the matrix-matrix multiplication.

```
status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB, csrB, SPARSE_STAGE_FINALIZE_MULT,
&csrC);
```

When the two-stage approach is not needed, you can perform both stages in a single call:

Single stage operation (`sparse_request_t = SPARSE_STAGE_FULL_MULT`)

```
status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB, csrB, SPARSE_STAGE_FULL_MULT, &csrC);
```

Matrix Manipulation Routines

The [Matrix Manipulation Routines](#) table lists the matrix manipulation routines and the data types associated with them.

Matrix Manipulation Routines and Their Data Types

Routine or Function Group	Data Types	Description
mkl_sparse_?_create_csr	s, d, c, z	Creates a handle for a CSR-format matrix.
mkl_sparse_?_create_csc	s, d, c, z	Creates a handle for a CSC format matrix.
mkl_sparse_?_create_coo	s, d, c, z	Creates a handle for a matrix in COO format.
mkl_sparse_?_create_bsr	s, d, c, z	Creates a handle for a matrix in BSR format.
mkl_sparse_copy	NA	Creates a copy of a matrix handle.
mkl_sparse_destroy	NA	Frees memory allocated for matrix handle.
mkl_sparse_convert_csr	NA	Converts internal matrix representation to CSR format.

Routine or Function Group	Data Types	Description
mkl_sparse_conver_t_bsr	NA	Converts internal matrix representation to BSR format or changes BSR block size.
mkl_sparse_?_export_csr	s, d, c, z	Exports CSR matrix from internal representation.
mkl_sparse_?_export_csc	s, d, c, z	Exports CSC matrix from internal representation.
mkl_sparse_?_export_bsr	s, d, c, z	Exports BSR matrix from internal representation.
mkl_sparse_?_set_value	s, d, c, z	Changes a single value of matrix in internal representation.
mkl_sparse_?_update_values	s, d, c, z	Changes all or selected matrix values in internal representation.
mkl_sparse_order	NA	Performs ordering of column indexes of the matrix in CSR format.

[mkl_sparse_?_create_csr](#)

Creates a handle for a CSR-format matrix.

Syntax

```
sparse_status_t mkl_sparse_s_create_csr (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, float *values);
```

```
sparse_status_t mkl_sparse_d_create_csr (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, double *values);
```

```
sparse_status_t mkl_sparse_c_create_csr (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, MKL_Complex8 *values);
```

```
sparse_status_t mkl_sparse_z_create_csr (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, MKL_Complex16 *values);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_?_create_csr` routine creates a handle for an m -by- k matrix A in CSR format.

NOTE

The input arrays provided are left unchanged except for the call to [mkl_sparse_order](#), which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use [mkl_sparse_copy](#).

Input Parameters

<i>indexing</i>	Indicates how input arrays are indexed.
<code>SPARSE_INDEX_BASE_ZERO</code>	Zero-based (C-style) indexing: indices start at 0.
<code>SPARSE_INDEX_BASE_ONE</code>	One-based (Fortran-style) indexing: indices start at 1.
<i>rows</i>	Number of rows of matrix A.
<i>cols</i>	Number of columns of matrix A.
<i>rows_start</i>	<p>Array of length at least <i>rows</i>. This array contains row indices, such that <i>rows_start</i>[<i>i</i>] - <i>indexing</i> is the first index of row <i>i</i> in the arrays <i>values</i> and <i>col_indx</i>. The value of <i>indexing</i> is 0 for zero-based indexing and 1 for one-based indexing.</p> <p>Refer to <i>pointerB</i> array description in CSR Format for more details.</p>
<i>rows_end</i>	<p>Array of at least length <i>rows</i>. This array contains row indices, such that <i>rows_end</i>[<i>i</i>] - <i>indexing</i> - 1 is the last index of row <i>i</i> in the arrays <i>values</i> and <i>col_indx</i>. The value of <i>indexing</i> is 0 for zero-based indexing and 1 for one-based indexing.</p> <p>Refer to <i>pointerE</i> array description in CSR Format for more details.</p>
<i>col_indx</i>	<p>For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix A. For zero-based indexing, array containing the column indices for each non-zero element of the matrix A. Its length is at least <i>rows_end</i>[<i>rows</i> - 1] - <i>indexing</i>.</p> <p>The value of <i>indexing</i> is 0 for zero-based indexing and 1 for one-based indexing.</p>
<i>values</i>	<p>Array containing non-zero elements of the matrix A. Its length is equal to length of the <i>col_indx</i> array.</p> <p>Refer to <i>values</i> array description in CSR Format for more details.</p>

Output Parameters

<i>A</i>	Handle containing internal data for subsequent Inspector-executor Sparse BLAS operations.
----------	---

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.

<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_?_create_csc

Creates a handle for a CSC format matrix.

Syntax

```

sparse_status_t mkl_sparse_s_create_csc (sparse_matrix_t *A, const sparse_index_base_t
indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *cols_start, MKL_INT
*cols_end, MKL_INT *row_indx, float *values);

sparse_status_t mkl_sparse_d_create_csc (sparse_matrix_t *A, const sparse_index_base_t
indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *cols_start, MKL_INT
*cols_end, MKL_INT *row_indx, double *values);

sparse_status_t mkl_sparse_c_create_csc (sparse_matrix_t *A, const sparse_index_base_t
indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *cols_start, MKL_INT
*cols_end, MKL_INT *row_indx, MKL_Complex8 *values);

sparse_status_t mkl_sparse_z_create_csc (sparse_matrix_t *A, const sparse_index_base_t
indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *cols_start, MKL_INT
*cols_end, MKL_INT *row_indx, MKL_Complex16 *values);

```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_?_create_csc` routine creates a handle for an m -by- k matrix A in CSC format.

NOTE

The input arrays provided are left unchanged except for the call to [mkl_sparse_order](#), which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use [mkl_sparse_copy](#).

Input Parameters

<i>indexing</i>	Indicates how input arrays are indexed.
	<code>SPARSE_INDEX_BASE_ZER</code> Zero-based (C-style) indexing: indices start at 0.
	<code>SPARSE_INDEX_BASE_ONE</code> One-based (Fortran-style) indexing: indices start at 1.
<i>rows</i>	Number of rows of the matrix A .
<i>cols</i>	Number of columns of the matrix A .
<i>cols_start</i>	Array of length at least m . This array contains col indices, such that <code>cols_start[i] - ind</code> is the first index of col i in the arrays <i>values</i> and <i>row_indx</i> . <i>ind</i> takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerB* array description in [CSC Format](#) for more details.

cols_end

Array of at least length *m*. This array contains col indices, such that *cols_end[i] - ind - 1* is the last index of col *i* in the arrays *values* and *row_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerE* array description in [CSC Format](#) for more details.

row_indx

For one-based indexing, array containing the row indices plus one for each non-zero element of the matrix *A*. For zero-based indexing, array containing the row indices for each non-zero element of the matrix *A*. Its length is at least *cols_end[cols - 1] - ind*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

values

Array containing non-zero elements of the matrix *A*. Its length is equal to length of the *row_indx* array.

Refer to *values* array description in [CSC Format](#) for more details.

Output Parameters

A

Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_create_coo

Creates a handle for a matrix in COO format.

Syntax

```
sparse_status_t mkl_sparse_s_create_coo (sparse_matrix_t *A, const sparse_index_base_t
indexing, const MKL_INT rows, const MKL_INT cols, const MKL_INT nnz, MKL_INT *row_indx,
MKL_INT * col_indx, float *values);

sparse_status_t mkl_sparse_d_create_coo (sparse_matrix_t *A, const sparse_index_base_t
indexing, const MKL_INT rows, const MKL_INT cols, const MKL_INT nnz, MKL_INT *row_indx,
MKL_INT * col_indx, double *values);

sparse_status_t mkl_sparse_c_create_coo (sparse_matrix_t *A, const sparse_index_base_t
indexing, const MKL_INT rows, const MKL_INT cols, const MKL_INT nnz, MKL_INT *row_indx,
MKL_INT * col_indx, MKL_Complex8 *values);
```

```
sparse_status_t mkl_sparse_z_create_coo (sparse_matrix_t *A, const sparse_index_base_t
indexing, const MKL_INT rows, const MKL_INT cols, const MKL_INT nnz, MKL_INT *row_indx,
MKL_INT * col_indx, MKL_Complex16 *values);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_?_create_coo` routine creates a handle for an m -by- k matrix A in COO format.

NOTE

The input arrays provided are left unchanged except for the call to [mkl_sparse_order](#), which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use [mkl_sparse_copy](#).

Input Parameters

<i>indexing</i>	Indicates how input arrays are indexed.				
	<table> <tr> <td><code>SPARSE_INDEX_BASE_ZERO</code></td><td>Zero-based (C-style) indexing: indices start at 0.</td></tr> <tr> <td><code>SPARSE_INDEX_BASE_ONE</code></td><td>One-based (Fortran-style) indexing: indices start at 1.</td></tr> </table>	<code>SPARSE_INDEX_BASE_ZERO</code>	Zero-based (C-style) indexing: indices start at 0.	<code>SPARSE_INDEX_BASE_ONE</code>	One-based (Fortran-style) indexing: indices start at 1.
<code>SPARSE_INDEX_BASE_ZERO</code>	Zero-based (C-style) indexing: indices start at 0.				
<code>SPARSE_INDEX_BASE_ONE</code>	One-based (Fortran-style) indexing: indices start at 1.				
<i>rows</i>	Number of rows of matrix A .				
<i>cols</i>	Number of columns of matrix A .				
<i>nnz</i>	Specifies the number of non-zero elements of the matrix A . Refer to <i>nnz</i> description in Coordinate Format for more details.				
<i>row_indx</i>	Array of length <i>nnz</i> , containing the row indices for each non-zero element of matrix A . Refer to <i>rows</i> array description in Coordinate Format for more details.				
<i>col_indx</i>	Array of length <i>nnz</i> , containing the column indices for each non-zero element of matrix A . Refer to <i>columns</i> array description in Coordinate Format for more details.				
<i>values</i>	Array of length <i>nnz</i> , containing the non-zero elements of matrix A in arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.				

Output Parameters

<i>A</i>	Handle containing internal data.
----------	----------------------------------

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
------------------------------------	-------------------------------

<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_?_create_bsr`

Creates a handle for a matrix in BSR format.

Syntax

```
sparse_status_t mkl_sparse_s_create_bsr (sparse_matrix_t *A, const sparse_index_base_t
indexing, const sparse_layout_t block_layout, const MKL_INT rows, const MKL_INT cols,
const MKL_INT block_size, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx,
float *values);
```

```
sparse_status_t mkl_sparse_d_create_bsr (sparse_matrix_t *A, const sparse_index_base_t
indexing, const sparse_layout_t block_layout, const MKL_INT rows, const MKL_INT cols,
const MKL_INT block_size, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx,
double *values);
```

```
sparse_status_t mkl_sparse_c_create_bsr (sparse_matrix_t *A, const sparse_index_base_t
indexing, const sparse_layout_t block_layout, const MKL_INT rows, const MKL_INT cols,
const MKL_INT block_size, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx,
MKL_Complex8 *values);
```

```
sparse_status_t mkl_sparse_z_create_bsr (sparse_matrix_t *A, const sparse_index_base_t
indexing, const sparse_layout_t block_layout, const MKL_INT rows, const MKL_INT cols,
const MKL_INT block_size, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx,
MKL_Complex16 *values);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_?_create_bsr` routine creates a handle for an m -by- k matrix A in BSR format.

NOTE

The input arrays provided are left unchanged except for the call to [mkl_sparse_order](#), which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use [mkl_sparse_copy](#).

Input Parameters

<i>indexing</i>	Indicates how input arrays are indexed.
<code>SPARSE_INDEX_BASE_ZERO</code>	Zero-based (C-style) indexing: indices start at 0.

	SPARSE_INDEX_BASE_ONE	One-based (Fortran-style) indexing: indices start at 1.
<i>block_layout</i>	Specifies layout of blocks:	
	SPARSE_LAYOUT_ROW_MAJOR	Storage of elements of blocks uses row major layout.
	SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements of blocks uses column major layout.
<i>rows</i>	Number of block rows of matrix A.	
<i>cols</i>	Number of block columns of matrix A.	
<i>block_size</i>	Size of blocks in matrix A.	
<i>rows_start</i>	Array of length <i>m</i> . This array contains row indices, such that <i>rows_start</i> [<i>i</i>] - <i>ind</i> is the first index of block row <i>i</i> in the arrays <i>values</i> and <i>col_indx</i> . <i>ind</i> takes 0 for zero-based indexing and 1 for one-based indexing.	
	Refer to <i>pointerB</i> array description in CSR Format for more details.	
<i>rows_end</i>	Array of length <i>m</i> . This array contains row indices, such that <i>rows_end</i> [<i>i</i>] - <i>ind</i> - 1 is the last index of block row <i>i</i> in the arrays <i>values</i> and <i>col_indx</i> . <i>ind</i> takes 0 for zero-based indexing and 1 for one-based indexing.	
	Refer to <i>pointerE</i> array description in CSR Format for more details.	
<i>col_indx</i>	For one-based indexing, array containing the column indices plus one for each non-zero block of the matrix A. For zero-based indexing, array containing the column indices for each non-zero block of the matrix A. Its length is <i>rows_end</i> [<i>rows</i> - 1] - <i>ind</i> . <i>ind</i> takes 0 for zero-based indexing and 1 for one-based indexing.	
<i>values</i>	Array containing non-zero elements of the matrix A. Its length is equal to length of the <i>col_indx</i> array multiplied by <i>block_size</i> * <i>block_size</i> .	
	Refer to the <i>values</i> array description in BSR Format for more details.	

Output Parameters

A	Handle containing internal data.
---	----------------------------------

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.

<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_copy

Creates a copy of a matrix handle.

Syntax

```
sparse_status_t mkl_sparse_copy (const sparse_matrix_t source, const struct
matrix_descr descr, sparse_matrix_t *dest);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_copy` routine creates a copy of a matrix handle.

NOTE

Currently, the `mkl_sparse_copy` routine does not support the descriptor argument and creates an exact (deep) copy of the input matrix.

Input Parameters

<i>source</i>	Specifies handle containing internal data.
<i>descr</i>	Structure specifying sparse matrix properties.
<code>sparse_matrix_type_t</code> <i>type</i>	- Specifies the type of a sparse matrix:
<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.
<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).
<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.
<code>sparse_fill_mode_t</code> <i>mode</i>	- Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWE The lower triangular matrix part is processed.
R

SPARSE_FILL_MODE_UPPE The upper triangular matrix part is processed.
R

sparse_diag_type_t *diag* - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT Diagonal elements might not be equal to one.

SPARSE_DIAG_UNIT Diagonal elements are equal to one.

Output Parameters

dest Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED Execution failed.

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

mkl_sparse_destroy

Frees memory allocated for matrix handle.

Syntax

```
sparse_status_t mkl_sparse_destroy (sparse_matrix_t A);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_destroy` routine frees memory allocated for matrix handle.

NOTE

You must free memory allocated for matrices after completing use of them. The `mkl_sparse_destroy` routine provides a utility to do so.

Input Parameters

A Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_convert_csr`

Converts internal matrix representation to CSR format.

Syntax

```
sparse_status_t mkl_sparse_convert_csr (const sparse_matrix_t source, const
sparse_operation_t operation, sparse_matrix_t *dest);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_convert_csr` routine converts internal matrix representation to CSR format.

When the source matrix is in COO format, the routine performs a sum reduction on duplicate elements.

Input Parameters

<i>source</i>	Handle containing internal data.
<i>operation</i>	Specifies operation <code>op()</code> on input matrix.
<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $op(A) = A$.
<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $op(A) = A^T$.
<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $op(A) = A^H$.

Output Parameters

<i>dest</i>	Handle containing internal data.
-------------	----------------------------------

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
------------------------------------	-------------------------------

<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_convert_bsr

Converts internal matrix representation to BSR format or changes BSR block size.

Syntax

```
sparse_status_t mkl_sparse_convert_bsr (const sparse_matrix_t source, const MKL_INT
block_size, const sparse_layout_t block_layout, const sparse_operation_t operation,
sparse_matrix_t *dest);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_convert_bsr` routine converts internal matrix representation to BSR format or changes BSR block size.

When the source matrix is in COO format, the routine performs a sum reduction on duplicate elements.

Input Parameters

<i>source</i>	Handle containing internal data.						
<i>block_size</i>	Size of the block in the output structure.						
<i>block_layout</i>	Specifies layout of blocks: <table> <tr> <td><code>SPARSE_LAYOUT_ROW_MAJOR</code></td><td>Storage of elements of blocks uses row major layout.</td></tr> <tr> <td><code>SPARSE_LAYOUT_COLUMN_MAJOR</code></td><td>Storage of elements of blocks uses column major layout.</td></tr> </table>	<code>SPARSE_LAYOUT_ROW_MAJOR</code>	Storage of elements of blocks uses row major layout.	<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Storage of elements of blocks uses column major layout.		
<code>SPARSE_LAYOUT_ROW_MAJOR</code>	Storage of elements of blocks uses row major layout.						
<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Storage of elements of blocks uses column major layout.						
<i>operation</i>	Specifies operation <code>op()</code> on input matrix. <table> <tr> <td><code>SPARSE_OPERATION_NON_TRANSPOSE</code></td><td>Non-transpose, $op(A) = A$.</td></tr> <tr> <td><code>SPARSE_OPERATION_TRANSPOSE</code></td><td>Transpose, $op(A) = A^T$.</td></tr> <tr> <td><code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code></td><td>Conjugate transpose, $op(A) = A^H$.</td></tr> </table>	<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $op(A) = A$.	<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $op(A) = A^T$.	<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $op(A) = A^H$.
<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $op(A) = A$.						
<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $op(A) = A^T$.						
<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $op(A) = A^H$.						

Output Parameters

dest

Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_?_export_csr`

Exports CSR matrix from internal representation.

Syntax

```
sparse_status_t mkl_sparse_s_export_csr (const sparse_matrix_t source,
sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **rows_start,
MKL_INT **rows_end, MKL_INT **col_indx, float **values);

sparse_status_t mkl_sparse_d_export_csr (const sparse_matrix_t source,
sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **rows_start,
MKL_INT **rows_end, MKL_INT **col_indx, double **values);

sparse_status_t mkl_sparse_c_export_csr (const sparse_matrix_t source,
sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **rows_start,
MKL_INT **rows_end, MKL_INT **col_indx, MKL_Complex8 **values);

sparse_status_t mkl_sparse_z_export_csr (const sparse_matrix_t source,
sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **rows_start,
MKL_INT **rows_end, MKL_INT **col_indx, MKL_Complex16 **values);
```

Include Files

- `mkl_spblas.h`

Description

If the matrix specified by the *source* handle is in CSR format, the `mkl_sparse_?_export_csr` routine exports an m -by- k matrix A in CSR format matrix from the internal representation. The routine returns pointers to the internal representation and does not allocate additional memory.

If the matrix is not already in CSR format, the routine returns `SPARSE_STATUS_INVALID_VALUE`.

Input Parameters

source

Handle containing internal data.

Output Parameters

<i>indexing</i>	Indicates how input arrays are indexed.
<code>SPARSE_INDEX_BASE_ZERO</code>	Zero-based (C-style) indexing: indices start at 0.
<code>SPARSE_INDEX_BASE_ONE</code>	One-based (Fortran-style) indexing: indices start at 1.
<i>rows</i>	Number of rows of the matrix <i>source</i> .
<i>cols</i>	Number of columns of the matrix <i>source</i> .
<i>rows_start</i>	<p>Pointer to array of length <i>m</i>. This array contains row indices, such that <i>rows_start</i>[<i>i</i>] - <i>ind</i> is the first index of row <i>i</i> in the arrays <i>values</i> and <i>col_indx</i>. <i>ind</i> takes 0 for zero-based indexing and 1 for one-based indexing.</p> <p>Refer to <i>pointerB</i> array description in CSR Format for more details.</p>
<i>rows_end</i>	<p>Pointer to array of length <i>m</i>. This array contains row indices, such that <i>rows_end</i>[<i>i</i>] - <i>ind</i> - 1 is the last index of row <i>i</i> in the arrays <i>values</i> and <i>col_indx</i>. <i>ind</i> takes 0 for zero-based indexing and 1 for one-based indexing.</p> <p>Refer to <i>pointerE</i> array description in CSR Format for more details.</p>
<i>col_indx</i>	<p>For one-based indexing, pointer to array containing the column indices plus one for each non-zero element of the matrix <i>source</i>. For zero-based indexing, pointer to array containing the column indices for each non-zero element of the matrix <i>source</i>. Its length is <i>rows_end</i>[<i>rows</i> - 1] - <i>ind</i>. <i>ind</i> takes 0 for zero-based indexing and 1 for one-based indexing.</p>
<i>values</i>	<p>Pointer to array containing non-zero elements of the matrix <i>A</i>. Its length is equal to length of the <i>col_indx</i> array.</p> <p>Refer to <i>values</i> array description in CSR Format for more details.</p>

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse?_export_csc`

Exports CSC matrix from internal representation.

Syntax

```
sparse_status_t mkl_sparse_s_export_csc (const sparse_matrix_t source,
sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **cols_start,
MKL_INT **cols_end, MKL_INT **row_indx, float **values);
```

```
sparse_status_t mkl_sparse_d_export_csc (const sparse_matrix_t source,
sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **cols_start,
MKL_INT **cols_end, MKL_INT **row_indx, double **values);
```

```
sparse_status_t mkl_sparse_c_export_csc (const sparse_matrix_t source,
sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **cols_start,
MKL_INT **cols_end, MKL_INT **row_indx, MKL_Complex8 **values);
```

```
sparse_status_t mkl_sparse_z_export_csc (const sparse_matrix_t source,
sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **cols_start,
MKL_INT **cols_end, MKL_INT **row_indx, MKL_Complex16 **values);
```

Include Files

- mkl_spblas.h

Description

If the matrix specified by the *source* handle is in CSC format, the `mkl_sparse_?_export_csc` routine exports an m -by- k matrix A in CSC format matrix from the internal representation. The routine returns pointers to the internal representation and does not allocate additional memory.

If the matrix is not already in CSC format, the routine returns `SPARSE_STATUS_INVALID_VALUE`.

Input Parameters

source Handle containing internal data.

Output Parameters

indexing Indicates how input arrays are indexed.

`SPARSE_INDEX_BASE_ZERO` Zero-based (C-style) indexing: indices start at 0.

`SPARSE_INDEX_BASE_ONE` One-based (Fortran-style) indexing: indices start at 1.

rows Number of rows of the matrix *source*.

cols Number of columns of the matrix *source*.

cols_start Array of length m . This array contains column indices, such that $cols_start[i] - cols_start[0]$ is the first index of column i in the arrays *values* and *row_indx*.

Refer to *pointerb* array description in [csc Format](#) for more details.

cols_end Pointer to array of length m . This array contains row indices, such that $cols_end[i] - cols_start[0] - 1$ is the last index of column i in the arrays *values* and *row_indx*.

Refer to *pointerE* array description in [csc Format](#) for more details.

<code>row_indx</code>	For one-based indexing, pointer to array containing the row indices plus one for each non-zero element of the matrix <i>source</i> . For zero-based indexing, pointer to array containing the row indices for each non-zero element of the matrix <i>source</i> . Its length is <code>cols_end[cols - 1] - cols_start[0]</code> .
<code>values</code>	Pointer to array containing non-zero elements of the matrix A. Its length is equal to length of the <code>row_indx</code> array. Refer to <code>values</code> array description in csc Format for more details.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_?_export_bsr`

Exports BSR matrix from internal representation.

Syntax

```
sparse_status_t mkl_sparse_s_export_bsr (const sparse_matrix_t source,
sparse_index_base_t *indexing, sparse_layout_t *block_layout, MKL_INT *rows, MKL_INT
*cols, MKL_INT *block_size, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT
**col_indx, float **values);

sparse_status_t mkl_sparse_d_export_bsr (const sparse_matrix_t source,
sparse_index_base_t *indexing, sparse_layout_t *block_layout, MKL_INT *rows, MKL_INT
*cols, MKL_INT *block_size, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT
**col_indx, double **values);

sparse_status_t mkl_sparse_c_export_bsr (const sparse_matrix_t source,
sparse_index_base_t *indexing, sparse_layout_t *block_layout, MKL_INT *rows, MKL_INT
*cols, MKL_INT *block_size, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT
**col_indx, MKL_Complex8 **values);

sparse_status_t mkl_sparse_z_export_bsr (const sparse_matrix_t source,
sparse_index_base_t *indexing, sparse_layout_t *block_layout, MKL_INT *rows, MKL_INT
*cols, MKL_INT *block_size, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT
**col_indx, MKL_Complex16 **values);
```

Include Files

- `mkl_splblas.h`

Description

If the matrix specified by the *source* handle is in BSR format, the `mkl_sparse_?_export_bsr` routine exports an $(block_size * rows)$ -by- $(block_size * cols)$ matrix *A* in BSR format from the internal representation. The routine returns pointers to the internal representation and does not allocate additional memory.

If the matrix is not already in BSR format, the routine returns `SPARSE_STATUS_INVALID_VALUE`.

Input Parameters

source Handle containing internal data.

Output Parameters

indexing Indicates how input arrays are indexed.

`SPARSE_INDEX_BASE_ZERO` Zero-based (C-style) indexing: indices start at 0.

`SPARSE_INDEX_BASE_ONE` One-based (Fortran-style) indexing: indices start at 1.

block_layout Specifies layout of blocks:

`SPARSE_LAYOUT_ROW_MAJOR` OR `SPARSE_LAYOUT_COLUMN_MAJOR` Storage of elements of blocks uses row major layout.

`SPARSE_LAYOUT_COLUMN_MAJOR` Storage of elements of blocks uses column major layout.

rows Number of block rows of the matrix *source*.

cols Number of block columns of matrix *source*.

block_size Size of the square block in matrix *source*.

rows_start Pointer to array of length *rows*. This array contains row indices, such that *rows_start*[*i*] - *ind* is the first index of block row *i* in the arrays *values* and *col_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerB* array description in [BSR Format](#) for more details.

rows_end Pointer to array of length *rows*. This array contains row indices, such that *rows_end*[*i*] - *ind* - 1 is the last index of block row *i* in the arrays *values* and *col_indx*. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to *pointerE* array description in [BSR Format](#) for more details.

col_indx For one-based indexing, pointer to array containing the column indices plus one for each non-zero blocks of the matrix *source*. For zero-based indexing, pointer to array containing the column indices for each non-zero blocks of the matrix *source*. Its length is *rows_end*[*rows* - 1] - *ind*[0]. *ind* takes 0 for zero-based indexing and 1 for one-based indexing.

values Pointer to array containing non-zero elements of matrix *source*. Its length is equal to length of the *col_indx* array multiplied by *block_size*block_size*.

Refer to the *values* array description in [BSR Format](#) for more details.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_set_value

Changes a single value of matrix in internal representation.

Syntax

```
sparse_status_t mkl_sparse_s_set_value (const sparse_matrix_t A, const MKL_INT row,
const MKL_INT col, const float value);

sparse_status_t mkl_sparse_d_set_value (const sparse_matrix_t A, const MKL_INT row,
const MKL_INT col, const double value);

sparse_status_t mkl_sparse_c_set_value (const sparse_matrix_t A, const MKL_INT row,
const MKL_INT col, const MKL_Complex8 value);

sparse_status_t mkl_sparse_z_set_value (const sparse_matrix_t A, const MKL_INT row,
const MKL_INT col, const MKL_Complex16 value);
```

Include Files

- mkl_spblas.h

Description

Use the `mkl_sparse_?_set_value` routine to change a single value of a matrix in the internal Inspector-executor Sparse BLAS format. The value should already be presented in a matrix structure.

Input Parameters

<i>A</i>	Specifies handle containing internal data.
<i>row</i>	Indicates row of matrix in which to set value.
<i>col</i>	Indicates column of matrix in which to set value.
<i>value</i>	Indicates value

Output Parameters

A Handle containing modified internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.

mkl_sparse_?_update_values

Changes all or selected matrix values in internal representation.

Syntax

NOTE

This routine is supported for sparse matrices in BSR format only.

```
sparse_status_t mkl_sparse_s_update_values (sparse_matrix_t A, MKL_INT nvalues, MKL_INT
*indx, MKL_INT *indy, float *values);
```

```
sparse_status_t mkl_sparse_d_update_values (sparse_matrix_t A, MKL_INT nvalues, MKL_INT
*indx, MKL_INT *indy, double *values);
```

```
sparse_status_t mkl_sparse_c_update_values (sparse_matrix_t A, MKL_INT nvalues, MKL_INT
*indx, MKL_INT *indy, MKL_Complex8 *values);
```

```
sparse_status_t mkl_sparse_z_update_values (sparse_matrix_t A, MKL_INT nvalues, MKL_INT
*indx, MKL_INT *indy, MKL_Complex16 *values);
```

Include Files

- mkl_spblas.h

Description

Use the `mkl_sparse_?_update_values` routine to change all or selected values of a matrix in the internal Inspector-Executor Sparse BLAS format.

The values to be updated should already be present in the matrix structure.

- To change selected values, you must provide an array `values` (with new values) and also the corresponding row and column indices for each value via `indx` and `indy` arrays as well as the overall number of changed elements `nvalues`.

So that, for example, to change `A(0, 0)` to 1 and `A(0, 1)` to 2, pass the following input parameters: `nvalues = 2`, `indx = {0, 0}`, `indy = {0, 1}` and `values = {1, 2}`.

- To change all the values in the matrix, provide the `values` array and explicitly set `nvalues` to 0 or the actual number of non zero elements. There is no need to supply `indx` and `indy` arrays.

Input Parameters

<i>A</i>	Specifies handle containing internal data.
<i>nvalues</i>	Total number of elements changed.
<i>indx</i>	Row indices for the new values.

NOTE

Currently, only updating the full matrix is supported. Set *indx* and *indy* as NULL.

<i>indy</i>	Column indices for the new values.
-------------	------------------------------------

NOTE

Currently, only updating the full matrix is supported. Set *indx* and *indy* as NULL.

<i>values</i>	New values.
---------------	-------------

Output Parameters

<i>A</i>	Handle containing modified internal data.
----------	---

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_order

Performs ordering of column indexes of the matrix in CSR format

Syntax

```
sparse_status_t mkl_sparse_order (const sparse_matrix_t csrA);
```

Include Files

- `mkl_spblas.h`

Description

Use the `mkl_sparse_order` routine to perform ordering of column indexes of the matrix in CSR format.

Input Parameters

`csrA` CSR data

Output Parameters

`csrA` Handle containing modified internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.

Inspector-Executor Sparse BLAS Analysis Routines

Analysis Routines and Their Data Types

Routine or Function Group	Description
mkl_sparse_set_lu_smoother_hint	Provides and estimate of the number and type of upcoming calls to LU smoother functionality.
mkl_sparse_set_mv_hint	Provides estimate of number and type of upcoming matrix-vector operations.
mkl_sparse_set_sv_hint	Provides estimate of number and type of upcoming triangular system solver operations.
mkl_sparse_set_mm_hint	Provides estimate of number and type of upcoming matrix-matrix multiplication operations.
mkl_sparse_set_sm_hint	Provides estimate of number and type of upcoming triangular matrix solve with multiple right hand sides operations.
mkl_sparse_set_dotmv_hint	Sets estimate of the number and type of upcoming matrix-vector operations.
mkl_sparse_set_symgs_hint	Sets estimate of number and type of upcoming <code>mkl_sparse_?_symgs</code> operations.
mkl_sparse_set_sorv_hint	Sets estimate of number and type of upcoming <code>mkl_sparse_?_symgs</code> operations.
mkl_sparse_set_memory_hint	Provides memory requirements for performance optimization purposes.
mkl_sparse_optimize	Analyzes matrix structure and performs optimizations using the hints provided in the handle.

Product and Performance Information

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

Notice revision #20201201

mkl_sparse_set_lu_smoother_hint

Provides an estimate of the number and type of upcoming calls to LU smoother functionality.

Syntax

```
sparse_status_t mkl_sparse_set_lu_smoother_hint (sparse_matrix_t A, const
sparse_operation_t operation, struct matrix_descr descr, MKL_INT expected_calls);
```

Include Files

- mkl_spblas.h

Description

The `mkl_sparse_set_lu_smoother_hint` function provides subsequent Inspector-Executor Sparse BLAS calls an estimate of the number of upcoming calls to the `lu_smoother` routine that ultimately may influence the optimizations applied and specifies whether or not to perform an operation on the matrix.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

operation

Specifies the operation *op()* on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE Non-transpose, $op(A) = A$.

SPARSE_OPERATION_TRANSPOSE Transpose, $op(A) = A^T$.

SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, $op(A) = A^H$.

descr

Structure specifying sparse matrix properties.

sparse_matrix_type_t type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL The matrix is processed as is.

SPARSE_MATRIX_TYPE_SYMMETRIC The matrix is symmetric (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_HERMITIAN The matrix is Hermitian (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_TRIANGULAR The matrix is triangular (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_DIAGONAL The matrix is diagonal (only diagonal elements are processed).

SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR The matrix is block-triangular (only the requested triangle is processed). Applies to BSR format only.

`SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL` The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`sparse_fill_mode_t`*mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

`SPARSE_FILL_MODE_LOWER` The lower triangular matrix part is processed.

`SPARSE_FILL_MODE_UPPER` The upper triangular matrix part is processed.

`sparse_diag_type_t`*diag* - Specifies the diagonal type for non-general matrices:

`SPARSE_DIAG_NON_UNIT` Diagonal elements might not be equal to one.

`SPARSE_DIAG_UNIT` Diagonal elements are equal to one.

`expected_calls`

Number of expected calls to execution routine.

`A`

Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_set_mv_hint`

Provides estimate of number and type of upcoming matrix-vector operations.

Syntax

```
sparse_status_t mkl_sparse_set_mv_hint (const sparse_matrix_t A, const
sparse_operation_t operation, const struct matrix_descr descr, const MKL_INT
expected_calls);
```

Include Files

- `mkl_splblas.h`

Description

Use the `mkl_sparse_set_mv_hint` routine to provide the Inspector-executor Sparse BLAS API an estimate of the number of upcoming matrix-vector multiplication operations for performance optimization, and specify whether or not to perform an operation on the matrix.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

operation

Specifies operation `op()` on input matrix.

`SPARSE_OPERATION_NON_TRANSPOSE` Non-transpose, $\text{op}(A) = A$.

`SPARSE_OPERATION_TRANSPOSE` Transpose, $\text{op}(A) = A^T$.

`SPARSE_OPERATION_CONJUGATE_TRANSPOSE` Conjugate transpose, $\text{op}(A) = A^H$.

descr

Structure specifying sparse matrix properties.

`sparse_matrix_type_t` *type* - Specifies the type of a sparse matrix:

`SPARSE_MATRIX_TYPE_GENERAL` The matrix is processed as is.

`SPARSE_MATRIX_TYPE_SYMMETRIC` The matrix is symmetric (only the requested triangle is processed).

`SPARSE_MATRIX_TYPE_HERMITIAN` The matrix is Hermitian (only the requested triangle is processed).

`SPARSE_MATRIX_TYPE_TRIANGULAR` The matrix is triangular (only the requested triangle is processed).

`SPARSE_MATRIX_TYPE_DIAGONAL` The matrix is diagonal (only diagonal elements are processed).

`SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR` The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.

`SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL` The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`sparse_fill_mode_t` *mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

`SPARSE_FILL_MODE_LOWER` The lower triangular matrix part is processed.

`SPARSE_FILL_MODE_UPPER` The upper triangular matrix part is processed.

`sparse_diag_type_t` *diag* - Specifies diagonal type for non-general matrices:

`SPARSE_DIAG_NON_UNIT` Diagonal elements might not be equal to one.

	<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.
<code>expected_calls</code>		Number of expected calls to execution routine.

Output Parameters

<code>A</code>	Handle containing internal data.
----------------	----------------------------------

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_set_sv_hint`

Provides estimate of number and type of upcoming triangular system solver operations.

Syntax

```
sparse_status_t mkl_sparse_set_sv_hint (const sparse_matrix_t A, const
sparse_operation_t operation, const struct matrix_descr descr, const MKL_INT
expected_calls);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_set_sv_hint` routine provides an estimate of the number of upcoming triangular system solver operations and type of these operations for performance optimization.

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

Input Parameters

<code>operation</code>	Specifies operation <code>op()</code> on input matrix.
	<code>SPARSE_OPERATION_NON_TRANSPOSE</code> Non-transpose, $op(A) = A$.

SPARSE_OPERATION_TRANSPOSE Transpose, $\text{op}(A) = A^T$.
SPOSE

SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, $\text{op}(A) = A^H$.
UGATE_TRANSPOSE

descr

Structure specifying sparse matrix properties.

sparse_matrix_type_t *type* - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL The matrix is processed as is.
NERAL

SPARSE_MATRIX_TYPE_SYMMETRIC The matrix is symmetric (only the requested triangle is processed).
MMETRIC

SPARSE_MATRIX_TYPE_HERMITIAN The matrix is Hermitian (only the requested triangle is processed).
RMITIAN

SPARSE_MATRIX_TYPE_TRIANGULAR The matrix is triangular (only the requested triangle is processed).
IANGULAR

SPARSE_MATRIX_TYPE_DIAGONAL The matrix is diagonal (only diagonal elements are processed).
AGONAL

SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
OCK_TRIANGULAR

SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.
OCK_DIAGONAL

sparse_fill_mode_t *mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER The lower triangular matrix part is processed.
R

SPARSE_FILL_MODE_UPPER The upper triangular matrix part is processed.
R

sparse_diag_type_t *diag* - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT Diagonal elements might not be equal to one.

SPARSE_DIAG_UNIT Diagonal elements are equal to one.

expected_calls

Number of expected calls to execution routine.

Output Parameters

A

Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS The operation was successful.

<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_set_mm_hint

Provides estimate of number and type of upcoming matrix-matrix multiplication operations.

Syntax

```
sparse_status_t mkl_sparse_set_mm_hint (const sparse_matrix_t A, const
sparse_operation_t operation, const struct matrix_descr descr, const sparse_layout_t
layout, const MKL_INT dense_matrix_size, const MKL_INT expected_calls);
```

Include Files

- `mkl_splblas.h`

Description

The `mkl_sparse_set_mm_hint` routine provides an estimate of the number of upcoming matrix-matrix multiplication operations and type of these operations for performance optimization purposes.

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

Input Parameters

<i>operation</i>	Specifies operation <code>op()</code> on input matrix. <table><tr><td><code>SPARSE_OPERATION_NON_TRANSPOSE</code></td><td>Non-transpose, $op(A) = A$.</td></tr><tr><td><code>SPARSE_OPERATION_TRANSPOSE</code></td><td>Transpose, $op(A) = A^T$.</td></tr><tr><td><code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code></td><td>Conjugate transpose, $op(A) = A^H$.</td></tr></table>	<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $op(A) = A$.	<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $op(A) = A^T$.	<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $op(A) = A^H$.
<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $op(A) = A$.						
<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $op(A) = A^T$.						
<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $op(A) = A^H$.						
<i>descr</i>	Structure specifying sparse matrix properties. <code>sparse_matrix_type_t type</code> - Specifies the type of a sparse matrix: <table><tr><td><code>SPARSE_MATRIX_TYPE_GENERAL</code></td><td>The matrix is processed as is.</td></tr><tr><td><code>SPARSE_MATRIX_TYPE_SYMMETRIC</code></td><td>The matrix is symmetric (only the requested triangle is processed).</td></tr></table>	<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.	<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).		
<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.						
<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).						

<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).
<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`sparse_fill_mode_t` *mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix part is processed.
<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix part is processed.

`sparse_diag_type_t` *diag* - Specifies diagonal type for non-general matrices:

<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements might not be equal to one.
<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.

layout

Specifies layout of elements:

<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Storage of elements uses column major layout.
<code>SPARSE_LAYOUT_ROW_MAJOR</code>	Storage of elements uses row major layout.

dense_matrix_size

Number of columns in dense matrix.

expected_calls

Number of expected calls to execution routine.

Output Parameters

A

Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.

<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_set_sm_hint

Provides estimate of number and type of upcoming triangular matrix solve with multiple right hand sides operations.

Syntax

```
sparse_status_t mkl_sparse_set_sm_hint (const sparse_matrix_t A, const
sparse_operation_t operation, const struct matrix_descr descr, const sparse_layout_t
layout, const MKL_INT dense_matrix_size, const MKL_INT expected_calls);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_set_sm_hint` routine provides an estimate of the number of upcoming triangular matrix solve with multiple right hand sides operations and type of these operations for performance optimization purposes.

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

Input Parameters

<i>operation</i>	Specifies operation <code>op()</code> on input matrix. <table><tr><td><code>SPARSE_OPERATION_NON_TRANSPOSE</code></td><td>Non-transpose, $op(A) = A$.</td></tr><tr><td><code>SPARSE_OPERATION_TRANSPOSE</code></td><td>Transpose, $op(A) = A^T$.</td></tr><tr><td><code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code></td><td>Conjugate transpose, $op(A) = A^H$.</td></tr></table>	<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $op(A) = A$.	<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $op(A) = A^T$.	<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $op(A) = A^H$.
<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $op(A) = A$.						
<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $op(A) = A^T$.						
<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $op(A) = A^H$.						
<i>descr</i>	Structure specifying sparse matrix properties. <code>sparse_matrix_type_t type</code> - Specifies the type of a sparse matrix: <table><tr><td><code>SPARSE_MATRIX_TYPE_GENERAL</code></td><td>The matrix is processed as is.</td></tr><tr><td><code>SPARSE_MATRIX_TYPE_SYMMETRIC</code></td><td>The matrix is symmetric (only the requested triangle is processed).</td></tr><tr><td><code>SPARSE_MATRIX_TYPE_HERMITIAN</code></td><td>The matrix is Hermitian (only the requested triangle is processed).</td></tr></table>	<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.	<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).	<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.						
<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).						
<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).						

<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).
<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`sparse_fill_mode_t` *mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix part is processed.
<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix part is processed.

`sparse_diag_type_t` *diag* - Specifies diagonal type for non-general matrices:

<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements might not be equal to one.
<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.

layout

Specifies layout of elements:

<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Storage of elements uses column major layout.
<code>SPARSE_LAYOUT_ROW_MAJOR</code>	Storage of elements uses row major layout.

dense_matrix_size

Number of right-hand-side.

expected_calls

Number of expected calls to execution routine.

Output Parameters

A

Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED

The requested operation is not supported.

mkl_sparse_set_dotmv_hint

Sets estimate of the number and type of upcoming matrix-vector operations.

Syntax

```
sparse_status_t mkl_sparse_set_dotmv_hint (const sparse_matrix_t A, const
sparse_operation_t operation, const struct matrix_descr descr, const MKL_INT
expected_calls);
```

Include Files

- mkl_spblas.h

Description

Use the `mkl_sparse_set_dotmv_hint` routine to provide the Inspector-executor Sparse BLAS API an estimate of the number of upcoming matrix-vector multiplication operations for performance optimization, and specify whether or not to perform an operation on the matrix.

Input Parameters

<i>operation</i>	Specifies the operation performed on matrix <i>A</i> . If <i>operation</i> = SPARSE_OPERATION_NON_TRANSPOSE, $\text{op}(A) = A$. If <i>operation</i> = SPARSE_OPERATION_TRANSPOSE, $\text{op}(A) = A^T$. If <i>operation</i> = SPARSE_OPERATION_CONJUGATE_TRANSPOSE, $\text{op}(A) = A^H$.
<i>descr</i>	Structure specifying sparse matrix properties. <i>sparse_matrix_type_t type</i> - Specifies the type of a sparse matrix: SPARSE_MATRIX_TYPE_GENERAL The matrix is processed as is. SPARSE_MATRIX_TYPE_SYMMETRIC The matrix is symmetric (only the requested triangle is processed). SPARSE_MATRIX_TYPE_HERMITIAN The matrix is Hermitian (only the requested triangle is processed). SPARSE_MATRIX_TYPE_TRIANGULAR The matrix is triangular (only the requested triangle is processed). SPARSE_MATRIX_TYPE_DIAGONAL The matrix is diagonal (only diagonal elements are processed). SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only. SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`sparse_fill_mode_t` *mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

`SPARSE_FILL_MODE_LOWER` The lower triangular matrix part is processed.

`SPARSE_FILL_MODE_UPPER` The upper triangular matrix part is processed.

`sparse_diag_type_t` *diag* - Specifies diagonal type for non-general matrices:

`SPARSE_DIAG_NON_UNIT` Diagonal elements might not be equal to one.

`SPARSE_DIAG_UNIT` Diagonal elements are equal to one.

expected_calls

Expected number of calls to the execution routine.

Output Parameters

A

Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_set_symgs_hint`

Syntax

Sets estimate of number and type of upcoming `mkl_sparse_?_symgs` operations.

```
sparse_status_t mkl_sparse_set_symgs_hint (const sparse_matrix_t A, const
sparse_operation_t operation, const struct matrix_descr descr, const MKL_INT
expected_calls);
```

Include Files

- `mkl_spblas.h`

Description

Use the `mkl_sparse_set_symgs_hint` routine to provide the Inspector-executor Sparse BLAS API an estimate of the number of upcoming symmetric Gauss-Seidel preconditioner operations for performance optimization, and specify whether or not to perform an operation on the matrix.

Input Parameters

<i>operation</i>	Specifies the operation performed on matrix A . If <i>operation</i> = SPARSE_OPERATION_NON_TRANSPOSE, $\text{op}(A) = A$. If <i>operation</i> = SPARSE_OPERATION_TRANSPOSE, $\text{op}(A) = A^T$. If <i>operation</i> = SPARSE_OPERATION_CONJUGATE_TRANSPOSE, $\text{op}(A) = A^H$.
<i>descr</i>	Structure specifying sparse matrix properties. <i>sparse_matrix_type_t type</i> - Specifies the type of a sparse matrix: SPARSE_MATRIX_TYPE_GENERAL The matrix is processed as is. SPARSE_MATRIX_TYPE_SYMMETRIC The matrix is symmetric (only the requested triangle is processed). SPARSE_MATRIX_TYPE_HERMITIAN The matrix is Hermitian (only the requested triangle is processed). SPARSE_MATRIX_TYPE_TRIANGULAR The matrix is triangular (only the requested triangle is processed). SPARSE_MATRIX_TYPE_DIAGONAL The matrix is diagonal (only diagonal elements are processed). SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only. SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only. <i>sparse_fill_mode_t mode</i> - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices: SPARSE_FILL_MODE_LOWER The lower triangular matrix part is processed. SPARSE_FILL_MODE_UPPER The upper triangular matrix part is processed. <i>sparse_diag_type_t diag</i> - Specifies diagonal type for non-general matrices: SPARSE_DIAG_NON_UNIT Diagonal elements might not be equal to one. SPARSE_DIAG_UNIT Diagonal elements are equal to one.
<i>diag</i>	Specifies diagonal type for non-general matrices
<i>mode</i>	Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices.
<i>type</i>	Specifies the type of a sparse matrix.
<i>expected_calls</i>	Estimate of the number to the execution routine.

Output Parameters

`A` Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_set_sorv_hint`

Sets an estimate of the number and type of upcoming `mkl_sparse_?_sorv` operations.

Syntax

```
sparse_status_t mkl_sparse_set_sorv_hint(
    const sparse_sor_type_t type,
    const sparse_matrix_t A,
    const struct matrix_descr descr,
    const MKL_INT expected_calls
);
```

Include Files

- `mkl_splblas.h`

Description

Use the `mkl_sparse_set_sorv_hint` routine to provide the Inspector-Executor Sparse BLAS API an estimate of the number of upcoming forward/backward sweeps or symmetric SOR preconditioner operations for performance optimization.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

`type` Specifies the operation performed by the SORV preconditioner.

`SPARSE_SOR_FORWARD` Performs forward sweep as defined by:

$$(\omega * L + D) * x^1 = (D - \omega * D - \omega * U) * x^0 + \omega * b$$

SPARSE_SOR_BACKWARD

Performs backward sweep as defined by:

$$(\omega * U + D) * x^1 = (D - \omega * D - \omega * L) * x^0 + \omega * b$$

SPARSE_SOR_SYMMETRIC

Preconditioner matrix could be expressed as:

$$\frac{\omega}{2 - \omega} \left(\frac{1}{\omega} D + L \right) D^{-1} \left(\frac{1}{\omega} D + L \right)^T$$

descr

Structure specifying sparse matrix properties.

sparse_matrix_type_t
type

Specifies the type of a sparse matrix:

- SPARSE_MATRIX_TYPE_GENERAL
The matrix is processed as-is.
- SPARSE_MATRIX_TYPE_SYMMETRIC
The matrix is symmetric (only the requested triangle is processed).
- SPARSE_MATRIX_TYPE_HERMITIAN
The matrix is Hermitian (only the requested triangle is processed).
- SPARSE_MATRIX_TYPE_TRIANGULAR
The matrix is triangular (only the requested triangle is processed).
- SPARSE_MATRIX_TYPE_DIAGONAL
The matrix is diagonal (only diagonal elements are processed).
- SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR
The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
- SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL
The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

sparse_fill_mode_t
mode

Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

- SPARSE_FILL_MODE_LOWER
The lower triangular matrix part is processed.
- SPARSE_FILL_MODE_UPPER
The upper triangular matrix part is processed.

sparse_diag_type_t
diag

Specifies diagonal type for non-general matrices:

- SPARSE_DIAG_NON_UNIT

Diagonal elements might not be equal to one.

- SPARSE_DIAG_UNIT

Diagonal elements are equal to one.

`A` Handle containing internal data.

`expected_calls` Estimate of the number of calls to the execution routine.

Output Parameters

`A` Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

`mkl_sparse_set_memory_hint`

Provides memory requirements for performance optimization purposes.

Syntax

```
sparse_status_t mkl_sparse_set_memory_hint (const sparse_matrix_t A, const
sparse_memory_usage_t policy);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_set_memory_hint` routine allocates additional memory for further performance optimization purposes.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>policy</i>	Specify memory utilization policy for optimization routine using these types:	
	SPARSE_MEMORY_NONE	Routine can allocate memory only for auxiliary structures (such as for workload balancing); the amount of memory is proportional to vector size.
	SPARSE_MEMORY_AGGRESSIVE	Default. Routine can allocate memory up to the size of matrix <i>A</i> for converting into the appropriate sparse format.

Output Parameters

<i>A</i>	Handle containing internal data.
----------	----------------------------------

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_optimize

Analyzes matrix structure and performs optimizations using the hints provided in the handle.

Syntax

```
sparse_status_t mkl_sparse_optimize (sparse_matrix_t A);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_optimize` routine analyzes matrix structure and performs optimizations using the hints provided in the handle. Generally, specifying a higher number of expected operations allows for more aggressive and time consuming optimizations.

Product and Performance Information

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

Product and Performance Information

Notice revision #20201201

Input Parameters

A Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

Inspector-Executor Sparse BLAS Execution Routines

Execution Routines and Their Data Types

Routine or Function Group	Data Types	Description
mkl_sparse?_lu_smoother	s, d, c, z	Computes an action of a preconditioner which corresponds to the approximate matrix decomposition $A \approx (L+D)*E*(U+D)$ for the system $Ax = b$
mkl_sparse?_mv	s, d, c, z	Computes a sparse matrix-vector product.
mkl_sparse?_trsv	s, d, c, z	Solves a system of linear equations for a square sparse matrix.
mkl_sparse?_mm	s, d, c, z	Computes the product of a sparse matrix and a dense matrix and stores the result as a dense matrix.
mkl_sparse?_trsm	s, d, c, z	Solves a system of linear equations with multiple right-hand sides for a square sparse matrix.
mkl_sparse?_add	s, d, c, z	Computes the sum of two sparse matrices. The result is stored in a newly allocated sparse matrix.
mkl_sparse_spmv	s, d, c, z	Computes the product of two sparse matrices and stores the result in a newly allocated sparse matrix.
mkl_sparse?_spmm	s, d, c, z	Computes the product of two sparse matrices and stores the result as a dense matrix.
mkl_sparse_sp2m	s, d, c, z	Computes the product of two sparse matrices (support operations on both matrices) and stores the result in a newly allocated sparse matrix.

Routine or Function Group	Data Types	Description
mkl_sparse_?_sp2md	s, d, c, z	Computes the product of two sparse matrices (support operations on both matrices) and stores the result as a dense matrix.
mkl_sparse_sypr	s, d, c, z	Computes the symmetric product of three sparse matrices and stores the result in a newly allocated sparse matrix.
mkl_sparse_?_syprd	s, d, c, z	Computes the symmetric triple product of a sparse matrix and a dense matrix and stores the result as a dense matrix.
mkl_sparse_?_symgs	s, d, c, z	Computes an action of a symmetric Gauss-Seidel preconditioner.
mkl_sparse_?_symgs_mv	s, d, c, z	Computes an action of a symmetric Gauss-Seidel preconditioner followed by a matrix-vector multiplication at the end.
mkl_sparse_?_syrgd	s, d, c, z	Computes the product of sparse matrix with its transpose (or conjugate transpose) and stores the result as a dense matrix.
mkl_sparse_syrgk	s, d, c, z	Computes the product of a sparse matrix with its transpose (or conjugate transpose) and stores the result in a newly allocated sparse matrix.
mkl_sparse_?_dotmv	s, d, c, z	Computes a sparse matrix-vector product followed by a dot product.

[mkl_sparse_?_lu_smoother](#)

Computes an action of a preconditioner which corresponds to the approximate matrix decomposition

$A \approx (L + D) \times E \times (U + D)$ for the system $Ax = b$ (see description below).

Syntax

```
sparse_status_t mkl_sparse_s_lu_smoother (const sparse_operation_t op, const
sparse_matrix_t A, const struct matrix_descr descr, const float *diag, const float
*approx_diag_inverse, float *x, const float *b);
```

```
sparse_status_t mkl_sparse_d_lu_smoother (const sparse_operation_t op, const
sparse_matrix_t A, const struct matrix_descr descr, const double *diag, const double
*approx_diag_inverse, double *x, const double *b);
```

```
sparse_status_t mkl_sparse_c_lu_smoother (const sparse_operation_t op, const
sparse_matrix_t A, const struct matrix_descr descr, const MKL_COMPLEX8 *diag, const
MKL_COMPLEX8 *approx_diag_inverse, MKL_COMPLEX8 *x, const MKL_COMPLEX8 *b);
```

```
sparse_status_t mkl_sparse_z_lu_smoother (const sparse_operation_t op, const
sparse_matrix_t A, const struct matrix_descr descr, const MKL_COMPLEX16 *diag, const
MKL_COMPLEX16 *approx_diag_inverse, MKL_COMPLEX16 *x, const MKL_COMPLEX16 *b);
```

Include Files

- `mkl_spblas.h`

Description

This routine computes an update for an iterative solution x of the system $Ax=b$ by means of applying one iteration of an approximate preconditioner which is based on the following approximation:

$A \sim (L + D) * E * (U + D)$, where E is an approximate inverse of the diagonal (using exact inverse will result in Gauss-Seidel preconditioner), L and U are lower/upper triangular parts of A , D is the diagonal (block diagonal in case of BSR format) of A .

The `mk1_sparse_?_lu_smoother` routine performs these operations:

```
r = b - A*x      /* 1. Computes the residual */
(L + D)*E*(U + D)*dx = r    /* 2. Finds the update dx by solving the system */
y = x + dx       /* 3. Performs an update */
```

This is also equal to the Symmetric Gauss-Seidel operation in the case of a CSR format and 1x1 diagonal blocks:

```
(L + D)*x^1 = b - U*x    /* Lower solve for intermediate x^1 */
(U + D)*x = b - L*x^1    /* Upper solve */
```

NOTE

This routine is supported only for non-transpose operation, real data types, and CSR/BSR sparse formats. In a BSR format, both diagonal values and approximate diagonal inverse arrays should be passed explicitly. For CSR format, diagonal values should be passed explicitly.

Input Parameters

<i>operation</i>	Specifies the operation performed on matrix A .
<code>SPARSE_OPERATION_NON_TRANSPOSE</code> , <code>op(A) := A</code>	<div> NOTE Transpose and conjugate transpose (<code>SPARSE_OPERATION_TRANSPOSE</code> and <code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>) are not supported. </div>
	Non-transpose, <code>op(A) = A</code> .
<i>A</i>	Handle which contains the sparse matrix A .
<i>descr</i>	Structure specifying sparse matrix properties.
<code>sparse_matrix_type_ttype</code>	- Specifies the type of a sparse matrix:
<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.
<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).

<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).
<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only the requested triangle is processed). Applies to BSR format only.
<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.
<code>sparse_fill_mode_t</code> <i>mode</i> - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:	
<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix part is processed.
<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix part is processed.
<code>sparse_diag_type_t</code> <i>diag</i> - Specifies the diagonal type for non-general matrices:	
<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements might not be equal to one.
<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.

NOTEOnly `SPARSE_MATRIX_TYPE_GENERAL` is supported.

<i>diag</i>	Array of size at least m , where m is the number of rows (or $nrows * block_size * block_size$ in case of BSR format) of matrix A . The array <i>diag</i> must contain the diagonal values of matrix A .
<i>approx_diag_inverse</i>	Array of size at least m , where m is the number of rows (or the number of $rows * block_size * block_size$ in case of BSR format) of matrix A . The array <i>approx_diag_inverse</i> will be used as E , approximate inverse of the diagonal of the matrix A .
<i>x</i>	Array of size at least k , where k is the number of columns (or $columns * block_size$ in case of BSR format) of matrix A . On entry, the array <i>x</i> must contain the input vector.
<i>b</i>	Array of size at least m , where m is the number of rows (or $rows * block_size$ in case of BSR format) of matrix A . The array <i>b</i> must contain the values of the right-hand side of the system.

Output Parameters

<i>x</i>	Overwritten by the computed vector y .
----------	--

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_?_mv

Computes a sparse matrix- vector product.

Syntax

```
sparse_status_t mkl_sparse_s_mv (const sparse_operation_t operation, const float alpha,
const sparse_matrix_t A, const struct matrix_descr descr, const float *x, const float
beta, float *y);
```

```
sparse_status_t mkl_sparse_d_mv (const sparse_operation_t operation, const double
alpha, const sparse_matrix_t A, const struct matrix_descr descr, const double *x, const
double beta, double *y);
```

```
sparse_status_t mkl_sparse_c_mv (const sparse_operation_t operation, const MKL_Complex8
alpha, const sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex8 *x,
const MKL_Complex8 beta, MKL_Complex8 *y);
```

```
sparse_status_t mkl_sparse_z_mv (const sparse_operation_t operation, const
MKL_Complex16 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const
MKL_Complex16 *x, const MKL_Complex16 beta, MKL_Complex16 *y);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_?_mv` routine computes a sparse matrix-dense vector product defined as

$$y := \alpha * \text{op}(A) * x + \beta * y$$

where:

α and β are scalars, x and y are vectors, and A is a sparse matrix handle of a matrix with m rows and k columns, and op is a matrix modifier for matrix A .

Input Parameters

<i>operation</i>	Specifies operation <code>op()</code> on input matrix.
<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $\text{op}(A) = A$.
<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $\text{op}(A) = A^T$.

	<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $\text{op}(A) = A^H$.
<i>alpha</i>		Specifies the scalar <i>alpha</i> .
<i>A</i>		Handle which contains the input matrix <i>A</i> .
<i>descr</i>		Structure specifying sparse matrix properties.
	<code>sparse_matrix_type_t type</code>	- Specifies the type of a sparse matrix:
	<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.
	<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).
	<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).
	<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).
	<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).
	<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
	<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.
	<code>sparse_fill_mode_t mode</code>	- Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
	<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix part is processed.
	<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix part is processed.
	<code>sparse_diag_type_t diag</code>	- Specifies diagonal type for non-general matrices:
	<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements might not be equal to one.
	<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.
<i>x</i>		Array of size equal to the number of columns, <i>k</i> of <i>A</i> if <i>operation</i> = <code>SPARSE_OPERATION_NON_TRANSPOSE</code> and at least the number of rows, <i>m</i> , of <i>A</i> otherwise. On entry, the array must contain the vector <i>x</i> .
<i>beta</i>		Specifies the scalar <i>beta</i> .
<i>y</i>		Array with size at least <i>m</i> if <i>operation</i> = <code>SPARSE_OPERATION_NON_TRANSPOSE</code> and at least <i>k</i> otherwise. On entry, the array <i>y</i> must contain the vector <i>y</i> . Array of size equal to the

number of rows, m of A if `operation = SPARSE_OPERATION_NON_TRANSPOSE` and at least the number of columns, k , of A otherwise. On entry, the array y must contain the vector y .

Output Parameters

y Overwritten by the updated vector y .

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_?_trsv`

Solves a system of linear equations for a triangular sparse matrix.

Syntax

```
sparse_status_t mkl_sparse_s_trsv (const sparse_operation_t operation, const float
alpha, const sparse_matrix_t A, const struct matrix_descr descr, const float *x, float
*y);
```

```
sparse_status_t mkl_sparse_d_trsv (const sparse_operation_t operation, const double
alpha, const sparse_matrix_t A, const struct matrix_descr descr, const double *x,
double *y);
```

```
sparse_status_t mkl_sparse_c_trsv (const sparse_operation_t operation, const
MKL_Complex8 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const
MKL_Complex8 *x, MKL_Complex8 *y);
```

```
sparse_status_t mkl_sparse_z_trsv (const sparse_operation_t operation, const
MKL_Complex16 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const
MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_?_trsv` routine solves a system of linear equations for a matrix:

$$\text{op}(A) * y = \text{alpha} * x$$

where A is a triangular sparse matrix, op is a matrix modifier for matrix A , alpha is a scalar, and x and y are vectors.

NOTE

For sparse matrices in the BSR format, the supported combinations of (*indexing,block_layout*) are:

- (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
- (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

<i>operation</i>	Specifies operation <code>op()</code> on input matrix.
	SPARSE_OPERATION_NON_TRANSPOSE Non-transpose, $\text{op}(A) = A$. SPARSE_OPERATION_TRANSPOSE Transpose, $\text{op}(A) = A^T$. SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, $\text{op}(A) = A^H$.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>A</i>	Handle which contains the input matrix <i>A</i> .
<i>descr</i>	Structure specifying sparse matrix properties.
	<code>sparse_matrix_type_t</code> <i>type</i> - Specifies the type of a sparse matrix:
	SPARSE_MATRIX_TYPE_GENERAL The matrix is processed as is. SPARSE_MATRIX_TYPE_SYMMETRIC The matrix is symmetric (only the requested triangle is processed). SPARSE_MATRIX_TYPE_HERMITIAN The matrix is Hermitian (only the requested triangle is processed). SPARSE_MATRIX_TYPE_TRIANGULAR The matrix is triangular (only the requested triangle is processed). SPARSE_MATRIX_TYPE_DIAGONAL The matrix is diagonal (only diagonal elements are processed). SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only. SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.
	<code>sparse_fill_mode_t</code> <i>mode</i> - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
	SPARSE_FILL_MODE_LOWER The lower triangular matrix part is processed. SPARSE_FILL_MODE_UPPER The upper triangular matrix part is processed.

`sparse_diag_type_t` *diag* - Specifies diagonal type for non-general matrices:

`SPARSE_DIAG_NON_UNIT` Diagonal elements might not be equal to one.

`SPARSE_DIAG_UNIT` Diagonal elements are equal to one.

x

Array of size at least *m*, where *m* is the number of rows of matrix *A*. On entry, the array must contain the vector *x*.

Output Parameters

y

Array of size at least *m* containing the solution to the system of linear equations.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_?_mm`

Computes the product of a sparse matrix and a dense matrix and stores the result as a dense matrix.

Syntax

```
sparse_status_t mkl_sparse_s_mm (const sparse_operation_t operation, const float alpha,
const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t layout,
const float *B, const MKL_INT columns, const MKL_INT ldb, const float beta, float *C,
const MKL_INT ldc);
```

```
sparse_status_t mkl_sparse_d_mm (const sparse_operation_t operation, const double
alpha, const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t
layout, const double *B, const MKL_INT columns, const MKL_INT ldb, const double beta,
double *C, const MKL_INT ldc);
```

```
sparse_status_t mkl_sparse_c_mm (const sparse_operation_t operation, const MKL_Complex8
alpha, const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t
layout, const MKL_Complex8 *B, const MKL_INT columns, const MKL_INT ldb, const
MKL_Complex8 beta, MKL_Complex8 *C, const MKL_INT ldc);
```

```
sparse_status_t mkl_sparse_z_mm (const sparse_operation_t operation, const
MKL_Complex16 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const
sparse_layout_t layout, const MKL_Complex16 *B, const MKL_INT columns, const MKL_INT
ldb, const MKL_Complex16 beta, MKL_Complex16 *C, const MKL_INT ldc);
```

Include Files

- mkl_splblas.h

Description

The `mkl_sparse_?_mm` routine performs a matrix-matrix operation:

```
C := alpha*op(A)*B + beta*C
```

where *alpha* and *beta* are scalars, *A* is a sparse matrix, `op` is a matrix modifier for matrix *A*, and *B* and *C* are dense matrices.

The `mkl_sparse_?_mm` and `mkl_sparse_?_trsm` routines support these configurations:

	Column-major dense matrix: <i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJOR	Row-major dense matrix: <i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
0-based sparse matrix: SPARSE_INDEX_BASE_ZERO	CSR BSR: general non-transposed matrix multiplication only	All formats
1-based sparse matrix: SPARSE_INDEX_BASE_ONE	All formats	CSR BSR: general non-transposed matrix multiplication only

NOTE
For sparse matrices in the BSR format, the supported combinations of (*indexing*,*block_layout*) are:

- (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
- (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

<i>operation</i>	Specifies operation <code>op()</code> on input matrix. SPARSE_OPERATION_NON_TRANSPOSE Non-transpose, <code>op(A) = A</code> . SPARSE_OPERATION_TRANSPOSE Transpose, <code>op(A) = A^T</code> . SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, <code>op(A) = A^H</code> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>A</i>	Handle which contains the sparse matrix <i>A</i> .
<i>descr</i>	Structure specifying sparse matrix properties. sparse_matrix_type_t <i>type</i> - Specifies the type of a sparse matrix: SPARSE_MATRIX_TYPE_GENERAL The matrix is processed as is.

SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`sparse_fill_mode_t` *mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.

`sparse_diag_type_t` *diag* - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT	Diagonal elements might not be equal to one.
SPARSE_DIAG_UNIT	Diagonal elements are equal to one.

layout

Describes the storage scheme for the dense matrix:

SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements uses column major layout.
SPARSE_LAYOUT_ROW_MAJOR	Storage of elements uses row major layout.

B

Array of size at least *rows*cols*.

	<i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>B</i>)	<i>ldb</i>	If $\text{op}(A) = A$, number of columns in <i>A</i> If $\text{op}(A) = A^T$, number of rows in <i>A</i>
<i>cols</i> (number of columns in <i>B</i>)	<i>columns</i>	<i>ldb</i>

columns

Number of columns of matrix *C*.

ldb Specifies the leading dimension of matrix *B*.

beta Specifies the scalar *beta*

C Array of size at least *rows*cols*, where

	<i>layout</i> = SPARSE_LAYOUT_COLU MN_MAJOR	<i>layout</i> = SPARSE_LAYOUT_ROW_MA JOR
<i>rows</i> (number of rows in <i>C</i>)	<i>ldc</i>	If $\text{op}(A) = A$, number of rows in <i>A</i> If $\text{op}(A) = A^T$, number of columns in <i>A</i>
<i>cols</i> (number of columns in <i>C</i>)	<i>columns</i>	<i>ldc</i>

ldc Specifies the leading dimension of matrix *C*.

Output Parameters

C Overwritten by the updated matrix *C*.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_trsm

Solves a system of linear equations with multiple right hand sides for a triangular sparse matrix.

Syntax

```
sparse_status_t mkl_sparse_s_trsm (const sparse_operation_t operation, const float
alpha, const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t
layout, const float *x, const MKL_INT columns, const MKL_INT ldx, float *y, const
MKL_INT ldy);
```

```
sparse_status_t mkl_sparse_d_trsm (const sparse_operation_t operation, const double
alpha, const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t
layout, const double *x, const MKL_INT columns, const MKL_INT ldx, double *y, const
MKL_INT ldy);
```

```

sparse_status_t mkl_sparse_c_trsm (const sparse_operation_t operation, const
MKL_Complex8 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const
sparse_layout_t layout, const MKL_Complex8 *x, const MKL_INT columns, const MKL_INT
ldx, MKL_Complex8 *y, const MKL_INT ldy);

sparse_status_t mkl_sparse_z_trsm (const sparse_operation_t operation, const
MKL_Complex16 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const
sparse_layout_t layout, const MKL_Complex16 *x, const MKL_INT columns, const MKL_INT
ldx, MKL_Complex16 *y, const MKL_INT ldy);

```

Include Files

- mkl_spblas.h

Description

The `mkl_sparse_?_trsm` routine solves a system of linear equations with multiple right hand sides for a triangular sparse matrix:

$$Y := \alpha * \text{inv}(\text{op}(A)) * X$$

where:

α is a scalar, X and Y are dense matrices, A is a sparse matrix, and op is a matrix modifier for matrix A .

The `mkl_sparse_?_mm` and `mkl_sparse_?_trsm` routines support these configurations:

	Column-major dense matrix: $\text{layout} =$ <code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Row-major dense matrix: layout $=$ <code>SPARSE_LAYOUT_ROW_MAJOR</code>
0-based sparse matrix: <code>SPARSE_INDEX_BASE_ZERO</code>	CSR BSR: general non-transposed matrix multiplication only	All formats
1-based sparse matrix: <code>SPARSE_INDEX_BASE_ONE</code>	All formats	CSR BSR: general non-transposed matrix multiplication only

NOTE

For sparse matrices in the BSR format, the supported combinations of ($\text{indexing}, \text{block_layout}$) are:

- (`SPARSE_INDEX_BASE_ZERO`, `SPARSE_LAYOUT_ROW_MAJOR`)
- (`SPARSE_INDEX_BASE_ONE`, `SPARSE_LAYOUT_COLUMN_MAJOR`)

Input Parameters

<i>operation</i>	Specifies operation $\text{op}()$ on input matrix.
<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $\text{op}(A) = A$.
<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $\text{op}(A) = A^T$.
<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $\text{op}(A) = A^H$.

<i>alpha</i>	Specifies the scalar <i>alpha</i> .														
<i>A</i>	Handle which contains the sparse matrix <i>A</i> .														
<i>descr</i>	Structure specifying sparse matrix properties.														
	<code>sparse_matrix_type_t</code> <i>type</i> - Specifies the type of a sparse matrix:														
	<table> <tr> <td><code>SPARSE_MATRIX_TYPE_GENERAL</code></td><td>The matrix is processed as is.</td></tr> <tr> <td><code>SPARSE_MATRIX_TYPE_SYMMETRIC</code></td><td>The matrix is symmetric (only the requested triangle is processed).</td></tr> <tr> <td><code>SPARSE_MATRIX_TYPE_HERMITIAN</code></td><td>The matrix is Hermitian (only the requested triangle is processed).</td></tr> <tr> <td><code>SPARSE_MATRIX_TYPE_TRIANGULAR</code></td><td>The matrix is triangular (only the requested triangle is processed).</td></tr> <tr> <td><code>SPARSE_MATRIX_TYPE_DIAGONAL</code></td><td>The matrix is diagonal (only diagonal elements are processed).</td></tr> <tr> <td><code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code></td><td>The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.</td></tr> <tr> <td><code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code></td><td>The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.</td></tr> </table>	<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.	<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).	<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).	<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).	<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).	<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.	<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.
<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.														
<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).														
<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).														
<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).														
<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).														
<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.														
<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.														
	<code>sparse_fill_mode_t</code> <i>mode</i> - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:														
	<table> <tr> <td><code>SPARSE_FILL_MODE_LOWER</code></td><td>The lower triangular matrix part is processed.</td></tr> <tr> <td><code>SPARSE_FILL_MODE_UPPER</code></td><td>The upper triangular matrix part is processed.</td></tr> </table>	<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix part is processed.	<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix part is processed.										
<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix part is processed.														
<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix part is processed.														
	<code>sparse_diag_type_t</code> <i>diag</i> - Specifies diagonal type for non-general matrices:														
	<table> <tr> <td><code>SPARSE_DIAG_NON_UNIT</code></td><td>Diagonal elements might not be equal to one.</td></tr> <tr> <td><code>SPARSE_DIAG_UNIT</code></td><td>Diagonal elements are equal to one.</td></tr> </table>	<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements might not be equal to one.	<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.										
<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements might not be equal to one.														
<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.														
<i>layout</i>	Describes the storage scheme for the dense matrix:														
	<table> <tr> <td><code>SPARSE_LAYOUT_COLUMN_MAJOR</code></td><td>Storage of elements uses column major layout.</td></tr> <tr> <td><code>SPARSE_LAYOUT_ROW_MAJOR</code></td><td>Storage of elements uses row major layout.</td></tr> </table>	<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Storage of elements uses column major layout.	<code>SPARSE_LAYOUT_ROW_MAJOR</code>	Storage of elements uses row major layout.										
<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Storage of elements uses column major layout.														
<code>SPARSE_LAYOUT_ROW_MAJOR</code>	Storage of elements uses row major layout.														
<i>x</i>	Array of size at least <i>rows*cols</i> .														

<i>layout</i> =	<i>layout</i> =
<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	<code>SPARSE_LAYOUT_ROW_MAJOR</code>

<i>rows</i> (number of rows in <i>x</i>)	<i>ldx</i>	number of rows in A
<i>cols</i> (number of columns in <i>x</i>)	<i>columns</i>	<i>ldx</i>

On entry, the array *x* must contain the matrix *X*.

columns

Number of columns in matrix *Y*.

ldx

Specifies the leading dimension of matrix *X*.

y

Array of size at least *rows*cols*, where

	<i>layout</i> = SPARSE_LAYOUT_COLUMN_MAJOR	<i>layout</i> = SPARSE_LAYOUT_ROW_MAJOR
<i>rows</i> (number of rows in <i>y</i>)	<i>ldy</i>	number of rows in A
<i>cols</i> (number of columns in <i>y</i>)	<i>columns</i>	<i>ldy</i>

Output Parameters

y

Overwritten by the updated matrix *Y*.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_add

Computes the sum of two sparse matrices. The result is stored in a newly allocated sparse matrix.

Syntax

```
sparse_status_t mkl_sparse_s_add (const sparse_operation_t operation, const
sparse_matrix_t A, const float alpha, const sparse_matrix_t B, sparse_matrix_t *C);

sparse_status_t mkl_sparse_d_add (const sparse_operation_t operation, const
sparse_matrix_t A, const double alpha, const sparse_matrix_t B, sparse_matrix_t *C);
```



```
sparse_status_t mkl_sparse_c_add (const sparse_operation_t operation, const
sparse_matrix_t A, const MKL_Complex8 alpha, const sparse_matrix_t B, sparse_matrix_t
*C);
```

```
sparse_status_t mkl_sparse_z_add (const sparse_operation_t operation, const
sparse_matrix_t A, const MKL_Complex16 alpha, const sparse_matrix_t B, sparse_matrix_t
*C);
```

Include Files

- mkl_spblas.h

Description

The `mkl_sparse_?_add` routine performs a matrix-matrix operation:

$$C := \alpha * \text{op}(A) + B$$

where *alpha* is a scalar, *op* is a matrix modifier, and *A*, *B*, and *C* are sparse matrices.

NOTE

This routine is only supported for sparse matrices in CSR and BSR formats. It is not supported for COO or CSC formats.

Input Parameters

<i>A</i>	Handle which contains the sparse matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>operation</i>	Specifies operation <code>op()</code> on input matrix. <div> <div>SPARSE_OPERATION_NON_TRANSPOSE</div> <div>Non-transpose, $\text{op}(A) = A$.</div> </div> <div> <div>SPARSE_OPERATION_TRANSPOSE</div> <div>Transpose, $\text{op}(A) = A^T$.</div> </div> <div> <div>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</div> <div>Conjugate transpose, $\text{op}(A) = A^H$.</div> </div>
<i>B</i>	Handle which contains the sparse matrix <i>B</i> .

Output Parameters

<i>C</i>	Handle which contains the resulting sparse matrix.
----------	--

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.

<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_spm

Computes the product of two sparse matrices. The result is stored in a newly allocated sparse matrix.

Syntax

```
sparse_status_t mkl_sparse_spm (const sparse_operation_t operation, const
sparse_matrix_t A, const sparse_matrix_t B, sparse_matrix_t *C);
```

Include Files

- `mkl_spmblas.h`

Description

The `mkl_sparse_spm` routine performs a matrix-matrix operation:

```
C := op(A) * B
```

where *A*, *B*, and *C* are sparse matrices and *op* is a matrix modifier for matrix *A*.

Notes

- This routine is supported only for sparse matrices in CSC, CSR, and BSR formats. It is not supported for sparse matrices in COO format.
 - The column indices of the output matrix (if in CSR format) can appear unsorted due to the algorithm chosen internally. To ensure sorted column indices (if that is important), call [mkl_sparse_order\(\)](#).
-

Input Parameters

<i>operation</i>	Specifies operation <code>op()</code> on input matrix. <div> <code>SPARSE_OPERATION_NON_TRANSPOSE</code> Non-transpose, $\text{op}(A) = A$. <code>SPARSE_OPERATION_TRANSPOSE</code> Transpose, $\text{op}(A) = A^T$. <code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code> Conjugate transpose, $\text{op}(A) = A^H$. </div>
<i>A</i>	Handle which contains the sparse matrix <i>A</i> .
<i>B</i>	Handle which contains the sparse matrix <i>B</i> .

Output Parameters

<i>C</i>	Handle which contains the resulting sparse matrix.
----------	--

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_?_spmmd

Computes the product of two sparse matrices and stores the result as a dense matrix.

Syntax

```
sparse_status_t mkl_sparse_s_spmmd (const sparse_operation_t operation, const
sparse_matrix_t A, const sparse_matrix_t B, const sparse_layout_t layout, float *C,
const MKL_INT ldc);

sparse_status_t mkl_sparse_d_spmmd (const sparse_operation_t operation, const
sparse_matrix_t A, const sparse_matrix_t B, const sparse_layout_t layout, double *C,
const MKL_INT ldc);

sparse_status_t mkl_sparse_c_spmmd (const sparse_operation_t operation, const
sparse_matrix_t A, const sparse_matrix_t B, const sparse_layout_t layout, MKL_Complex8
*C, const MKL_INT ldc);

sparse_status_t mkl_sparse_z_spmmd (const sparse_operation_t operation, const
sparse_matrix_t A, const sparse_matrix_t B, const sparse_layout_t layout, MKL_Complex16
*C, const MKL_INT ldc);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_?_spmmd` routine performs a matrix-matrix operation:

$$C := op(A) * B$$

where *A* and *B* are sparse matrices, *op* is a matrix modifier for matrix *A*, and *C* is a dense matrix.

NOTE

This routine is not supported for sparse matrices in the COO format. For sparse matrices in BSR format, these combinations of (*indexing*, *block_layout*) are supported:

- (`SPARSE_INDEX_BASE_ZERO`, `SPARSE_LAYOUT_ROW_MAJOR`)
- (`SPARSE_INDEX_BASE_ONE`, `SPARSE_LAYOUT_COLUMN_MAJOR`)

Input Parameters

operation Specifies operation `op()` on input matrix.

	SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, $\text{op}(A) = A$.
	SPARSE_OPERATION_TRANSPOSE	Transpose, $\text{op}(A) = A^T$.
	SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, $\text{op}(A) = A^H$.
<i>A</i>		Handle which contains the sparse matrix <i>A</i> .
<i>B</i>		Handle which contains the sparse matrix <i>B</i> .
<i>layout</i>		Describes the storage scheme for the dense matrix:
	SPARSE_LAYOUT_COLUMN_MAJOR	Storage of elements uses column major layout.
	SPARSE_LAYOUT_ROW_MAJOR	Storage of elements uses row major layout.
<i>ldC</i>		Leading dimension of matrix <i>C</i> .

Output Parameters

<i>C</i>	Resulting dense matrix.
----------	-------------------------

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_sp2m

Computes the product of two sparse matrices. The result is stored in a newly allocated sparse matrix.

Syntax

```
sparse_status_t mkl_sparse_sp2m (const sparse_operation_t transA, const struct
matrix_descr descrA, const sparse_matrix_t A, const sparse_operation_t transB, const
struct matrix_descr descrB, const sparse_matrix_t B, const sparse_request_t request,
sparse_matrix_t *C);
```

Include Files

- `mkl_splblas.h`

Description

The `mk1_sparse_sp2m` routine performs a matrix-matrix operation:

$$C := \text{opA}(A) * \text{opB}(B)$$

where A , B , and C are sparse matrices, opA and opB are matrix modifiers for matrices A and B , respectively.

NOTE

The column indices of the output matrix (if in CSR format) can appear unsorted due to the algorithm chosen internally. To ensure sorted column indices (if that is important), call [mk1_sparse_order\(\)](#).

Input Parameters

opA

Specifies operation on input matrix.

<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $\text{op}(A)=A$
<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $\text{op}(A)=A^T$
<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $\text{op}(A)=A^H$

opB

Specifies operation on input matrix.

<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose, $\text{op}(B)=B$
<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose, $\text{op}(B)=B^T$
<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose, $\text{op}(B)=B^H$

descrA

Structure that specifies sparse matrix properties.

NOTE Currently, only `SPARSE_MATRIX_TYPE_GENERAL` is supported.

`sparse_matrix_type_t` specifies the type of sparse matrix.

<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.
<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).

SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only the requested triangle is processed). This applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only the requested triangle is processed). This applies to BSR format only.

`sparse_fill_mode_t`*mode* specifies the triangular matrix portion for symmetric, Hermitian, triangular, and block-triangular matrices.

SPARSE_FILL_MODE_LOWER	The lower triangular matrix is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix is processed.

`sparse_diag_type_t`*diag* specifies the type of diagonal for non-general matrices.

SPARSE_DIAG_NON_UNIT	Diagonal elements must not be equal to 1.
SPARSE_DIAG_UNIT	Diagonal elements are equal to 1.

descrB

Structure that specifies sparse matrix properties.

NOTE Currently, only SPARSE_MATRIX_TYPE_GENERAL is supported.

`sparse_matrix_type_t`*type* specifies the type of sparse matrix.

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only the requested triangle is processed). This applies to BSR format only.

SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only the requested triangle is processed). This applies to BSR format only.
-----------------------------------	---

`sparse_fill_mode_t`*mode* specifies the triangular matrix portion for symmetric, Hermitian, triangular, and block-triangular matrices.

SPARSE_FILL_MODE_LOWER	The lower triangular matrix is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix is processed.

`sparse_diag_type_t`*diag* specifies the type of diagonal for non-general matrices.

SPARSE_DIAG_NON_UNIT	Diagonal elements must not be equal to 1.
SPARSE_DIAG_UNIT	Diagonal elements are equal to 1.

A

Handle which contains the sparse matrix *A*.

B

Handle which contains the sparse matrix *B*.

request

Specifies whether the full computations are performed at once or using the two-stage algorithm. See [Two-stage Algorithm for Inspector-executor Sparse BLAS Routines](#).

SPARSE_STAGE_NNZ_COUNT	Only <code>rowIndex</code> (BSR/CSR format) or <code>colIndex</code> (CSC format) array of the matrix is computed internally. The computation can be extracted to measure the memory required for full operation.
SPARSE_STAGE_FINALIZE_MULT_NO_VAL	Finalize computations of the matrix structure (values will not be computed). Use only after the call with <code>SPARSE_STAGE_NNZ_COUNT</code> parameter.
SPARSE_STAGE_FINALIZE_MULT	Finalize computation. Can also be used when the matrix structure remains unchanged and only values of the resulting matrix <i>C</i> need to be recomputed.
SPARSE_STAGE_FULL_MULT_NO_VAL	Perform computations of the matrix structure.
SPARSE_STAGE_FULL_MULT	Perform the entire computation in a single step.

Output Parameters

C

Handle which contains the resulting sparse matrix.

Return Values

The function returns a value indicating whether the operation was successful, or the reason why it failed.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	The internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	The execution failed.
D	
SPARSE_STATUS_INTERNAL_ERROR	An error occurred in the implementation of the algorithm.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_sp2md

Computes the product of two sparse matrices (support operations on both matrices) and stores the result as a dense matrix.

Syntax

```
sparse_status_t mkl_sparse_s_sp2md ( const sparse_operation_t transA, const struct
matrix_descr descrA, const sparse_matrix_t A, const sparse_operation_t transB, const
struct matrix_descr descrB, const sparse_matrix_t B, const float alpha, const float
beta, float *C, const sparse_layout_t layout, const MKL_INT ldc );

sparse_status_t mkl_sparse_d_sp2md ( const sparse_operation_t transA, const struct
matrix_descr descrA, const sparse_matrix_t A, const sparse_operation_t transB, const
struct matrix_descr descrB, const sparse_matrix_t B, const double alpha, const double
beta, double *C, const sparse_layout_t layout, const MKL_INT ldc );

sparse_status_t mkl_sparse_c_sp2md ( const sparse_operation_t transA, const struct
matrix_descr descrA, const sparse_matrix_t A, const sparse_operation_t transB, const
struct matrix_descr descrB, const sparse_matrix_t B, const MKL_Complex8 alpha, const
MKL_Complex8 beta, MKL_Complex8 *C, const sparse_layout_t layout, const MKL_INT ldc );

sparse_status_t mkl_sparse_z_sp2md ( const sparse_operation_t transA, const struct
matrix_descr descrA, const sparse_matrix_t A, const sparse_operation_t transB, const
struct matrix_descr descrB, const sparse_matrix_t B, const MKL_Complex16 alpha, const
MKL_Complex16 beta, MKL_Complex16 *C, const sparse_layout_t layout, const MKL_INT
ldc );
```

Include Files

- mkl_spblas.h

Description

The `mkl_sparse_?_sp2md` routine performs a matrix-matrix operation:

$$C = \alpha * opA(A) * opB(B) + \beta * C$$

where *A* and *B* are sparse matrices, *opA* is a matrix modifier for matrix *A*, *opB* is a matrix modifier for matrix *B*, and *C* is a dense matrix, *alpha* and *beta* are scalars.

NOTE

This routine is not supported for sparse matrices in the COO format. For sparse matrices in BSR format, these combinations of (indexing, block_layout) are supported:

- (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
- (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

transA

Specifies operation *op()* on the input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, $op(A)=A$
SPARSE_OPERATION_TRANSPOSE	Transpose, $op(A)=A^T$
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, $op(A)=A^H$

descrA

Structure that specifies the sparse matrix properties.

NOTE Currently, only SPARSE_MATRIX_TYPE_GENERAL is supported.

sparse_matrix_type_t specifies the type of sparse matrix.

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only the requested triangle is processed). This applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only the requested triangle is processed). This applies to BSR format only.

sparse_fill_mode_t specifies the triangular matrix portion for symmetric, Hermitian, triangular, and block-triangular matrices.

SPARSE_FILL_MODE_LOWER	The lower triangular matrix is processed.
------------------------	---

SPARSE_FILL_MODE_UPPER	The upper triangular matrix is processed.
------------------------	---

`sparse_diag_type_t`*diag* specifies the type of diagonal for non-general matrices.

SPARSE_DIAG_NON_UNIT	Diagonal elements must not be equal to 1.
SPARSE_DIAG_UNIT	Diagonal elements are equal to 1.

A

Handle which contains the sparse matrix *A*.

transB

Specifies operation `opB()` on the input matrix.

SPARSE_OPERATION_NON_TRANSPOSE	Non-transpose, $\text{opB}(B) = B$.
SPARSE_OPERATION_TRANSPOSE	Transpose, $\text{opB}(B) = B^T$.
SPARSE_OPERATION_CONJUGATE_TRANSPOSE	Conjugate transpose, $\text{opB}(B) = B^H$.

descrB

Structure that specifies the sparse matrix properties.

NOTE

Currently, only `SPARSE_MATRIX_TYPE_GENERAL` is supported.

`sparse_matrix_type_t`*type* specifies the type of sparse matrix.

SPARSE_MATRIX_TYPE_GENERAL	The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR	The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL	The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR	The matrix is block-triangular (only the requested triangle is processed). This applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL	The matrix is block-diagonal (only the requested triangle is processed). This applies to BSR format only.

`sparse_fill_mode_t`*mode* specifies the triangular matrix portion for symmetric, Hermitian, triangular, and block-triangular matrices.

<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix is processed.
<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix is processed.

`sparse_diag_type_t` *diag* specifies the type of diagonal for non-general matrices.

<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements must not be equal to 1.
<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to 1.

B Handle which contains the sparse matrix *B*.

alpha Specifies the scalar alpha.

beta Specifies the scalar beta.

layout Describes the storage scheme for the dense matrix:

<code>SPARSE_LAYOUT_COLUMN_MAJOR</code>	Storage of elements uses column major layout.
<code>SPARSE_LAYOUT_ROW_MAJOR</code>	Storage of elements uses row major layout.

ldc Leading dimension of matrix *C*.

Output Parameters

C The resulting dense matrix.

Return Values

The function returns a value indicating whether the operation was successful, or the reason why it failed.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	The internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	The execution failed.
<code>D</code>	
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error occurred in the implementation of the algorithm.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_sypr

Computes the symmetric product of three sparse matrices and stores the result in a newly allocated sparse matrix.

Syntax

```
sparse_status_t mkl_sparse_sypr (const sparse_operation_t operation , const
sparse_matrix_t A, const sparse_matrix_t B, const struct matrix_descr B,
sparse_matrix_t *C, const sparse_request_t request);
```

Include Files

- `mkl_splblas.h`

Description

The `mkl_sparse_sypr` routine performs a multiplication of three sparse matrices that results in a symmetric or Hermitian matrix, C .

```
C:=A*B*opA(A)
```

or

```
C:=opA(A) *B*A
```

depending on the matrix modifier *operation*.

Here, A , B , and C are sparse matrices, where A has a general structure while B and C are symmetric (for real data types) or Hermitian (for complex data types) matrices. `opA` is the transpose (real data types) or conjugate transpose (complex data types) operator.

NOTE

This routine is not supported for sparse matrices in COO or CSC formats. This routine supports only CSR and BSR formats. In addition, it supports only the sorted CSR and sorted BSR formats for the input matrix. If the data is unsorted, call the [mkl_sparse_order](#) routine before either [mkl_sparse_sypr](#) or [mkl_sparse_?_syprd](#).

Input Parameters

`operation`

Specifies operation on the input sparse matrices.

<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose case. $C:=A*B*(A^T)$ for real precision $C:=A*B*(A^H)$ for complex precision.
<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose case. This is not supported for complex matrices. $C:=(A^T)*B*A$
<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose case. This is not supported for real matrices. $C:=(A^H)*B*A$

`A`

Handle which contains the sparse matrix A .

B

Handle which contains the sparse matrix *B*.

descrB

Structure specifying properties of the sparse matrix.

`sparse_matrix_type_t` *type* specifies the type of a sparse matrix

SPARSE_MATRIX_TYPE_SYMMETRIC	The matrix is symmetric (only the specified triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN	The matrix is Hermitian (only the specified triangle is processed).

`sparse_fill_mode_t` *mode* specifies the triangular matrix part.

SPARSE_FILL_MODE_LOWER	The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.

`sparse_diag_type_t` *diag* specifies the type of diagonal.

SPARSE_DIAG_NON_UNIT	Diagonal elements cannot be equal to one.
----------------------	---

NOTEThis routine also supports $C=AA^{T,H}$ with these parameters:`descrB.type=SPARSE_MATRIX_TYPE_DIAGONAL``descrB.diag=SPARSE_DIAG_UNIT`In this case, you do not need to allocate structure B. Use the routine as a 2-stage version of [mkl_sparse_syrk](#).

request

Use this routine to specify if the computations should be performed in a single step or using the two-stage algorithm. See [Two-stage Algorithm for Inspector-executor Sparse BLAS Routines](#) for more information.

SPARSE_STAGE_NNZ_COUNT	Only <code>rowIndex</code> (BSR/CSR format) or <code>colIndex</code> (CSC format) array of the matrix is computed internally. The computation can be extracted to measure the memory required for full operation.
SPARSE_STAGE_FINALIZE_MULT_N_O_VAL	Finalize computations of the matrix structure (values will not be computed). Use only after the call with <code>SPARSE_STAGE_NNZ_COUNT</code> parameter.
SPARSE_STAGE_FINALIZE_MULT	Finalize computation. Can be used after the call with the <code>SPARSE_STAGE_NNZ_COUNT</code> or

	SPARSE_STAGE_FINALIZE_MULT_NO_VAL. Can also be used when the matrix structure remains unchanged and only values of the resulting matrix <i>C</i> need to be recomputed.
SPARSE_STAGE_FULL_MULT_NO_VAL	Perform computations of the matrix structure.
SPARSE_STAGE_FULL_MULT	Perform the entire computation in a single step.

Output Parameters

C Handle which contains the resulting sparse matrix. Only the upper-triangular part of the matrix is computed.

Return Values

The function returns a value indicating whether the operation was successful, or the reason why it failed.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_syprd

Computes the symmetric triple product of a sparse matrix and a dense matrix and stores the result as a dense matrix.

Syntax

```
sparse_status_t mkl_sparse_s_syprd (const sparse_operation_t op, const sparse_matrix_t
A, const float *B, const sparse_layout_t layoutB, const MKL_INT ldb, const float alpha,
const float beta, float *C, const sparse_layout_t layoutC, const MKL_INT ldc);

sparse_status_t mkl_sparse_d_syprd (const sparse_operation_t op, const sparse_matrix_t
A, const double *B, const sparse_layout_t layoutB, const MKL_INT ldb, const double
alpha, const double beta, double *C, const sparse_layout_t layoutC, const MKL_INT ldc);

sparse_status_t mkl_sparse_c_syprd (const sparse_operation_t op, const sparse_matrix_t
A, const MKL_Complex8 *B, const sparse_layout_t layoutB, const MKL_INT ldb, const
MKL_Complex8 alpha, const MKL_Complex8 beta, MKL_Complex8 *C, const sparse_layout_t
layoutC, const MKL_INT ldc);

sparse_status_t mkl_sparse_z_syprd (const sparse_operation_t op, const sparse_matrix_t
A, const MKL_Complex16 *B, const sparse_layout_t layoutB, const MKL_INT ldb, const
MKL_Complex16 alpha, const MKL_Complex16 beta, MKL_Complex16 *C, const sparse_layout_t
layoutC, const MKL_INT ldc);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_?_syprd` routine performs a multiplication of three sparse matrices that results in a symmetric or Hermitian matrix, C .

$$C := \alpha * A * B * \text{op}(A) + \beta * C$$

or

$$C := \alpha * \text{op}(A) * B * A + \beta * C$$

depending on the matrix modifier `operation`. Here A is a sparse matrix, B and C are dense and symmetric (or Hermitian) matrices.

`op` is the transpose (real precision) or conjugate transpose (complex precision) operator.

NOTE

This routine is not supported for sparse matrices in COO or CSC formats. It supports only CSR and BSR formats. In addition, this routine supports only the sorted CSR and sorted BSR formats for the input matrix. If the data is unsorted, call the [mkl_sparse_order](#) routine before either [mkl_sparse_sypr](#) or [mkl_sparse_?_syprd](#).

Input Parameters

`operation`

Specifies operation on the input sparse matrix.

<code>SPARSE_OPERATION_NON_TRANSPOSE</code>	Non-transpose case. $C := \alpha * A * B * (A^T)$ $+ \beta * C$ for real precision. $C := \alpha * A * B * (A^H)$ $+ \beta * C$ for complex precision.
<code>SPARSE_OPERATION_TRANSPOSE</code>	Transpose case. This is not supported for complex matrices. $C := \alpha * (A^T) * B * A$ $+ \beta * C$
<code>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</code>	Conjugate transpose case. This is not supported for real matrices. $C := \alpha * (A^H) * B * A$ $+ \beta * C$

`A`

Handle which contains the sparse matrix A .

`B`

Input dense matrix. Only the upper triangular part of the matrix is used for computation.

denselayoutB

Structure that describes the storage scheme for the dense matrix.

SPARSE_LAYOUT_COLUMN_MAJOR	Store elements in a column-major layout.
SPARSE_LAYOUT_ROW_MAJOR	Store elements in a row-major layout.

ldb

Leading dimension of matrix B.

alpha

Scalar parameter.

beta

Scalar parameter.

NOTE

Since the upper triangular part of matrix C is the only portion that is processed, set real values of `alpha` and `beta` in the complex case to obtain the Hermitian matrix.

denselayoutC

Structure that describes the storage scheme for the dense matrix.

SPARSE_LAYOUT_COLUMN_MAJOR	Store elements in a column-major layout.
SPARSE_LAYOUT_ROW_MAJOR	Store elements in a row-major layout.

ldc

Leading dimension of matrix C.

Output Parameters

C

Handle which contains the resulting dense matrix. Only the upper-triangular part of the matrix is computed.

Return Values

The function returns a value indicating whether the operation was successful, or the reason why it failed.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	The internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	The execution failed.
D	
SPARSE_STATUS_INTERNAL_ERROR	An error occurred in the implementation of the algorithm.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse?_symgs

Computes a symmetric Gauss-Seidel preconditioner.

Syntax

```
sparse_status_t mkl_sparse_s_symgs (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const float alpha, const float *b,
float *x);
```

```
sparse_status_t mkl_sparse_d_symgs (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const double alpha, const double
*b, double *x);
```

```
sparse_status_t mkl_sparse_c_symgs (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex8 alpha, const
MKL_Complex8 *b, MKL_Complex8 *x);
```

```
sparse_status_t mkl_sparse_z_symgs (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex16 alpha, const
MKL_Complex16 *b, MKL_Complex16 *x);
```

Include Files

- mkl_spblas.h

Description

The `mkl_sparse_?_symgs` routine performs this operation:

```
x0 := x*alpha;
(L + D)*x1 = b - U*x0;
(U + D)*x = b - L*x1;
```

where $A = L + D + U$.

NOTE

This routine is not supported for sparse matrices in BSR, COO, or CSC formats. It supports only the CSR format. Additionally, only symmetric matrices are supported, so the `desc.type` must be `SPARSE_MATRIX_TYPE_SYMMETRIC`.

Input Parameters

operation Specifies the operation performed on matrix *A*.
`SPARSE_OPERATION_NON_TRANSPOSE`, `op(A) := A`.

NOTE

Transpose (`SPARSE_OPERATION_TRANSPOSE`) and conjugate transpose (`SPARSE_OPERATION_CONJUGATE_TRANSPOSE`) are not supported.

A Handle which contains the sparse matrix *A*.

alpha Specifies the scalar *alpha*.

descr Structure specifying sparse matrix properties.

`sparse_matrix_type_t type` - Specifies the type of a sparse matrix:

<code>SPARSE_MATRIX_TYPE_GENERAL</code>	The matrix is processed as is.
<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code>	The matrix is symmetric (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_HERMITIAN</code>	The matrix is Hermitian (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code>	The matrix is triangular (only the requested triangle is processed).
<code>SPARSE_MATRIX_TYPE_DIAGONAL</code>	The matrix is diagonal (only diagonal elements are processed).
<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code>	The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code>	The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.
<code>sparse_fill_mode_t</code> <i>mode</i>	- Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
<code>SPARSE_FILL_MODE_LOWER</code>	The lower triangular matrix part is processed.
<code>SPARSE_FILL_MODE_UPPER</code>	The upper triangular matrix part is processed.
<code>sparse_diag_type_t</code> <i>diag</i>	- Specifies diagonal type for non-general matrices:
<code>SPARSE_DIAG_NON_UNIT</code>	Diagonal elements might not be equal to one.
<code>SPARSE_DIAG_UNIT</code>	Diagonal elements are equal to one.
<i>x</i>	Array of size at least <i>m</i> , where <i>m</i> is the number of rows of matrix <i>A</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .
<i>b</i>	Array of size at least <i>m</i> , where <i>m</i> is the number of rows of matrix <i>A</i> . On entry, the array <i>b</i> must contain the vector <i>b</i> .

Output Parameters

<i>x</i>	Overwritten by the computed vector <i>x</i> .
----------	---

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.

<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse?_symgs_mv

Computes a symmetric Gauss-Seidel preconditioner followed by a matrix-vector multiplication.

Syntax

```
sparse_status_t mkl_sparse_s_symgs_mv (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const float alpha, const float *b,
float *x, float *y);

sparse_status_t mkl_sparse_d_symgs_mv (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const double alpha, const double
*b, double *x, double *y);

sparse_status_t mkl_sparse_c_symgs_mv (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex8 alpha, const
MKL_Complex8 *b, MKL_Complex8 *x, MKL_Complex8 *y);

sparse_status_t mkl_sparse_z_symgs_mv (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex16 alpha, const
MKL_Complex16 *b, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse?_symgs_mv` routine performs this operation:

```
x0 := x*alpha;
(L + D)*x1 = b - U*x0;
(U + D)*x = b - L*x1;
y := A*x
```

where $A = L + D + U$

NOTE

This routine is not supported for sparse matrices in BSR, COO, or CSC formats. It supports only the CSR format. Additionally, only symmetric matrices are supported, so the `desc.type` must be `SPARSE_MATRIX_TYPE_SYMMETRIC`.

Input Parameters

operation Specifies the operation performed on input matrix.
`SPARSE_OPERATION_NON_TRANSPOSE`, $\text{op}(A) = A$.

NOTE

Transpose (`SPARSE_OPERATION_TRANSPOSE`) and conjugate transpose (`SPARSE_OPERATION_CONJUGATE_TRANSPOSE`) are not supported.

<i>A</i>	Handle which contains the sparse matrix <i>A</i> .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>descr</i>	Structure specifying sparse matrix properties.
	<code>sparse_matrix_type_t</code> <i>type</i> - Specifies the type of a sparse matrix:
	<code>SPARSE_MATRIX_TYPE_GENERAL</code> The matrix is processed as is.
	<code>SPARSE_MATRIX_TYPE_SYMMETRIC</code> The matrix is symmetric (only the requested triangle is processed).
	<code>SPARSE_MATRIX_TYPE_HERMITIAN</code> The matrix is Hermitian (only the requested triangle is processed).
	<code>SPARSE_MATRIX_TYPE_TRIANGULAR</code> The matrix is triangular (only the requested triangle is processed).
	<code>SPARSE_MATRIX_TYPE_DIAGONAL</code> The matrix is diagonal (only diagonal elements are processed).
	<code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code> The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
	<code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code> The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.
	<code>sparse_fill_mode_t</code> <i>mode</i> - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
	<code>SPARSE_FILL_MODE_LOWER</code> The lower triangular matrix part is processed.
	<code>SPARSE_FILL_MODE_UPPER</code> The upper triangular matrix part is processed.
	<code>sparse_diag_type_t</code> <i>diag</i> - Specifies diagonal type for non-general matrices:
	<code>SPARSE_DIAG_NON_UNIT</code> Diagonal elements might not be equal to one.
	<code>SPARSE_DIAG_UNIT</code> Diagonal elements are equal to one.
<i>x</i>	Array of size at least <i>m</i> , where <i>m</i> is the number of rows of matrix <i>A</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .
<i>b</i>	Array of size at least <i>m</i> , where <i>m</i> is the number of rows of matrix <i>A</i> . On entry, the array <i>b</i> must contain the vector <i>b</i> .

Output Parameters

x	Overwritten by the computed vector x .
y	Array of size at least m , where m is the number of rows of matrix A . Overwritten by the computed vector y .

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_syrk`

Computes the product of sparse matrix with its transpose (or conjugate transpose) and stores the result in a newly allocated sparse matrix.

Syntax

```
sparse_status_t mkl_sparse_syrk (const sparse_operation_t operation, const
sparse_matrix_t A, sparse_matrix_t *C);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_syrk` routine performs a sparse matrix-matrix operation which results in a sparse matrix C that is either Symmetric (real) or Hermitian (complex):

$$C := A * \text{op}(A)$$

where `op(*)` is the transpose for real matrices and conjugate transpose for complex matrices OR

$$C := \text{op}(A) * A$$

depending on the matrix modifier `op` which can be the transpose for real matrices or conjugate transpose for complex matrices.

Here, A and C are sparse matrices.

NOTE This routine is not supported for sparse matrices in COO or CSC formats. It supports only CSR and BSR formats. Additionally, this routine supports only the sorted CSR and sorted BSR formats for the input matrix. If data is unsorted, call the [mkl_sparse_order](#) routine before either `mkl_sparse_syrk` or `mkl_sparse_?_syrkd`.

Input Parameters

<i>operation</i>	Specifies the operation <code>op()</code> on input matrix . SPARSE_OPERATION_NON_TRANSPOSE, Non-transpose, $C := A * op(A)$ where <code>op(*)</code> is the transpose for real matrices and conjugate transpose for complex matrices SPARSE_OPERATION_TRANSPOSE, Transpose, $C := (A^T) * A$ for real matrix A SPARSE_OPERATION_CONJUGATE_TRANSPOSE, Conjugate transpose, $C := (A^H) * A$ for complex matrix A.
<i>A</i>	Handle which contains the sparse matrix A.

Output Parameters

<i>C</i>	Handle which contains the resulting sparse matrix. Only the upper-triangular part of the matrix is computed.
----------	--

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_syrkd

Computes the product of sparse matrix with its transpose (or conjugate transpose) and stores the result as a dense matrix.

Syntax

```
sparse_status_t mkl_sparse_s_syrkd (sparse_operation_t operation, const sparse_matrix_t
A, float alpha, float beta, float *C, sparse_layout_t layout, MKL_INT ldc);

sparse_status_t mkl_sparse_d_syrkd (sparse_operation_t operation, const sparse_matrix_t
A, double alpha, double beta, double *C, sparse_layout_t layout, MKL_INT ldc);

sparse_status_t mkl_sparse_c_syrkd (sparse_operation_t operation, const sparse_matrix_t
A, const MKL_Complex8 alpha, MKL_Complex8 beta, MKL_Complex8 *C, sparse_layout_t
layout, MKL_INT ldc);

sparse_status_t mkl_sparse_z_syrkd (sparse_operation_t operation, const sparse_matrix_t
A, MKL_Complex16 alpha, MKL_Complex16 beta, MKL_Complex16 *C, sparse_layout_t layout,
const MKL_INT ldc);
```

Include Files

- `mkl_splblas.h`

Description

The `mkl_sparse_?_syrkd` routine performs a sparse matrix-matrix operation which results in a dense matrix C that is either symmetric (real case) or Hermitian (complex case):

$$C := \text{beta} * C + \text{alpha} * \text{op}(A)$$

or

$$C := \text{beta} * C + \text{alpha} * \text{op}(A) * A$$

depending on the matrix modifier `op` which can be the transpose for real matrices or conjugate transpose for complex matrices. Here, A is a sparse matrix and C is a dense matrix.

NOTE This routine is not supported for sparse matrices in COO or CSC formats. It supports only CSR and BSR formats. Additionally, this routine supports only the sorted CSR and sorted BSR formats for the input matrix. If data is unsorted, call the [mkl_sparse_order](#) routine before either `mkl_sparse_syrk` or `mkl_sparse_?_syrkd`.

Input Parameters

<i>operation</i>	Specifies the operation <code>op()</code> performed on the input matrix. SPARSE_OPERATION_NON_TRANSPOSE, Non-transpose, $C := \text{beta} * C + \text{alpha} * A * \text{op}(A)$ where <code>op(*)</code> is the transpose (real matrices) or conjugate transpose (complex matrices). SPARSE_OPERATION_TRANSPOSE, Transpose, $C := \text{beta} * C + \text{alpha} * A^T * A$ for real matrix A . SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, $C := \text{beta} * C + \text{alpha} * A^H * A$ for complex matrix A .
<i>A</i>	Handle which contains the sparse matrix A .
<i>alpha</i>	Scalar parameter <i>alpha</i> .
<i>beta</i>	Scalar parameter <i>beta</i> .
<i>layout</i>	Describes the storage scheme for the dense matrix. <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> $\text{layout} = \text{SPARSE_LAYOUT_COLUMN_MAJOR}$ Storage of elements uses column-major layout. $\text{layout} = \text{SPARSE_LAYOUT_ROW_MAJOR}$ Storage of elements uses row-major layout. </div>
<i>ldc</i>	Leading dimension of matrix C .

NOTE

Only the upper triangular part of matrix C is processed. Therefore, you must set real values of *alpha* and *beta* for complex matrices in order to obtain a Hermitian matrix.

Output Parameters

`C` Resulting dense matrix. Only the upper triangular part of the matrix is computed.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_?_dotmv`

Computes a sparse matrix-vector product followed by a dot product.

Syntax

```
sparse_status_t mkl_sparse_s_dotmv (const sparse_operation_t operation, const float
alpha, const sparse_matrix_t A, const struct matrix_descr descr, const float *x, const
float beta, float *y, float *d);

sparse_status_t mkl_sparse_d_dotmv (const sparse_operation_t operation, const double
alpha, const sparse_matrix_t A, const struct matrix_descr descr, const double *x, const
double beta, double *y, double *d);

sparse_status_t mkl_sparse_c_dotmv (const sparse_operation_t operation, const
MKL_Complex8 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const
MKL_Complex8 *x, const MKL_Complex8 beta, MKL_Complex8 *y, MKL_Complex8 *d);

sparse_status_t mkl_sparse_z_dotmv (const sparse_operation_t operation, const
MKL_Complex16 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const
MKL_Complex16 *x, const MKL_Complex16 beta, MKL_Complex16 *y, MKL_Complex16 *d);
```

Include Files

- `mkl_spblas.h`

Description

The `mkl_sparse_?_dotmv` routine computes a sparse matrix-vector product and dot product:

$$y := \alpha \text{op}(A) * x + \beta y \quad d := \sum_i x_i * y_i \quad (\text{real case})$$

$$d := \sum_i \text{conj}(x_i) * y_i \quad (\text{complex case})$$

where

- α and β are scalars.
- x and y are vectors.
- A is an m -by- k matrix.

- *conj* represents complex conjugation.
- $\text{op}(A)$ is a matrix modifier.

Available options for $\text{op}(A)$ are A , A^T , or A^H .

NOTE

For sparse matrices in the BSR format, the supported combinations of (*indexing,block_layout*) are:

- (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
- (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

<i>operation</i>	Specifies the operation performed on matrix A . If <i>operation</i> = SPARSE_OPERATION_NON_TRANSPOSE, $\text{op}(A) = A$. If <i>operation</i> = SPARSE_OPERATION_TRANSPOSE, $\text{op}(A) = A^T$. If <i>operation</i> = SPARSE_OPERATION_CONJUGATE_TRANSPOSE, $\text{op}(A) = A^H$.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>A</i>	Handle which contains the sparse matrix A .
<i>descr</i>	Structure specifying sparse matrix properties.
	<i>sparse_matrix_type_t type</i> - Specifies the type of a sparse matrix:
	SPARSE_MATRIX_TYPE_GENERAL The matrix is processed as is.
	SPARSE_MATRIX_TYPE_SYMMETRIC The matrix is symmetric (only the requested triangle is processed).
	SPARSE_MATRIX_TYPE_HERMITIAN The matrix is Hermitian (only the requested triangle is processed).
	SPARSE_MATRIX_TYPE_TRIANGULAR The matrix is triangular (only the requested triangle is processed).
	SPARSE_MATRIX_TYPE_DIAGONAL The matrix is diagonal (only diagonal elements are processed).
	SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
	SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.
	<i>sparse_fill_mode_t mode</i> - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
	SPARSE_FILL_MODE_LOWER The lower triangular matrix part is processed.

	SPARSE_FILL_MODE_UPPER	The upper triangular matrix part is processed.
	R	
	sparse_diag_type_t <i>diag</i>	- Specifies diagonal type for non-general matrices:
	SPARSE_DIAG_NON_UNIT	Diagonal elements might not be equal to one.
	SPARSE_DIAG_UNIT	Diagonal elements are equal to one.
<i>x</i>	If <i>operation</i> = SPARSE_OPERATION_NON_TRANSPOSE, array of size at least <i>k</i> , where <i>k</i> is the number of columns of matrix <i>A</i> . Otherwise, array of size at least <i>m</i> , where <i>m</i> is the number of rows of matrix <i>A</i> . On entry, the array <i>x</i> must contain the vector <i>x</i> .	
<i>beta</i>	Specifies the scalar <i>beta</i> .	
<i>y</i>	If <i>operation</i> = SPARSE_OPERATION_NON_TRANSPOSE, array of size at least <i>m</i> , where <i>k</i> is the number of rows of matrix <i>A</i> . Otherwise, array of size at least <i>k</i> , where <i>k</i> is the number of columns of matrix <i>A</i> . On entry, the array <i>y</i> must contain the vector <i>y</i> .	

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
<i>d</i>	Overwritten by the dot product of <i>x</i> and <i>y</i> .

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS	The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED	The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED	Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE	The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED	Execution failed.
SPARSE_STATUS_INTERNAL_ERROR	An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED	The requested operation is not supported.

mkl_sparse_?_solv

Computes forward, backward sweeps or a symmetric successive over-relaxation preconditioner operation.

Syntax

```
sparse_status_t mkl_sparse_s_solv(
    const sparse_sor_type_t type,
    const struct matrix_descr descrA,
```

```
const sparse_matrix_t A,
float omega,
float alpha,
float* x,
float* b
);
```

```
sparse_status_t mkl_sparse_d_sorv(
const sparse_sor_type_t type,
const struct matrix_descr descrA,
const sparse_matrix_t A,
double omega,
double alpha,
double* x,
double* b
);
```

Include Files

- mkl_spblas.h

Description

The mkl_sparse_?_sorv routine performs one of the following operations:

SPARSE_SOR_FORWARD:

$$(\omega * L + D) * x^1 = (D - \omega * D - \omega * U) * x^0 + \omega * b$$

SPARSE_SOR_BACKWARD:

$$(\omega * U + D) * x^1 = (D - \omega * D - \omega * L) * x^0 + \omega * b$$

SPARSE_SOR_SYMMETRIC: Performs application of a

$$\frac{\omega}{2 - \omega} \left(\frac{1}{\omega} D + L \right) D^{-1} \left(\frac{1}{\omega} D + L \right)^T$$

preconditioner.

where $A = L + D + U$ and x^0 is an input vector x scaled by input parameter α vector and x^1 is an output stored in vector x .

NOTE

Currently this routine only supports the following configuration:

- CSR format of the input matrix
- SPARSE_SOR_FORWARD operation
- General matrix (`descr.type` is `SPARSE_MATRIX_TYPE_GENERAL`) or symmetric matrix with full portrait and unit diagonal (`descr.type` is `SPARSE_MATRIX_TYPE_SYMMETRIC`, `descr.mode` is `SPARSE_FILL_MODE_FULL`, and `descr.diag` is `SPARSE_DIAG_UNIT`)

NOTE

Currently, this routine is optimized only for sequential threading execution mode.

Warning It is currently not allowed to place a `sorv` call in a parallel section (e.g., under `#pragma omp parallel`), because it is not thread-safe in this scenario. This limitation will be addressed in one of the upcoming releases.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

type

Specifies the operation performed by the SORV preconditioner.

SPARSE_SOR_FORWARD Performs forward sweep as defined by:

$$(\omega * L + D) * x^1 = (D - \omega * D - \omega * U) * x^0 + \omega * b$$

SPARSE_SOR_BACKWARD Performs backward sweep as defined by:

$$(\omega * U + D) * x^1 = (D - \omega * D - \omega * L) * x^0 + \omega * b$$

SPARSE_SOR_SYMMETRIC Preconditioner matrix could be expressed as:

$$\frac{\omega}{2 - \omega} \left(\frac{1}{\omega} D + L \right) D^{-1} \left(\frac{1}{\omega} D + L \right)^T$$

descr

Structure specifying sparse matrix properties.

sparse_matrix_type_t Specifies the type of a sparse matrix:
 type

- SPARSE_MATRIX_TYPE_GENERAL
The matrix is processed as-is.
- SPARSE_MATRIX_TYPE_SYMMETRIC
The matrix is symmetric (only the requested triangle is processed).
- SPARSE_MATRIX_TYPE_HERMITIAN

	<p>The matrix is Hermitian (only the requested triangle is processed).</p> <ul style="list-style-type: none"> • <code>SPARSE_MATRIX_TYPE_TRIANGULAR</code> <p>The matrix is triangular (only the requested triangle is processed).</p> <ul style="list-style-type: none"> • <code>SPARSE_MATRIX_TYPE_DIAGONAL</code> <p>The matrix is diagonal (only diagonal elements are processed).</p> <ul style="list-style-type: none"> • <code>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</code> <p>The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.</p> <ul style="list-style-type: none"> • <code>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</code> <p>The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.</p>
<code>sparse_fill_mode_t</code> mode	<p>Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:</p> <ul style="list-style-type: none"> • <code>SPARSE_FILL_MODE_LOWER</code> <p>The lower triangular matrix part is processed.</p> <ul style="list-style-type: none"> • <code>SPARSE_FILL_MODE_UPPER</code> <p>The upper triangular matrix part is processed.</p>
<code>sparse_diag_type_t</code> diag	<p>Specifies diagonal type for non-general matrices:</p> <ul style="list-style-type: none"> • <code>SPARSE_DIAG_NON_UNIT</code> <p>Diagonal elements might not be equal to one.</p> <ul style="list-style-type: none"> • <code>SPARSE_DIAG_UNIT</code> <p>Diagonal elements are equal to one.</p>
<code>A</code>	Handle containing internal data.
<code>omega</code>	Relaxation factor.
<code>alpha</code>	Parameter that could be used to normalize or set to zero the vector <code>x</code> that holds the initial guess.
<code>x</code>	Initial guess on input.
<code>b</code>	Right-hand side.
Output Parameters	
<code>x</code>	Solution vector on output.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

BLAS-like Extensions

Intel® oneAPI Math Kernel Library provides C and Fortran routines to extend the functionality of the BLAS routines. These include routines to compute vector products, matrix-vector products, and matrix-matrix products.

Intel® oneAPI Math Kernel Library also provides routines to perform certain data manipulation, including matrix in-place and out-of-place transposition operations combined with simple matrix arithmetic operations. Transposition operations are Copy As Is, Conjugate transpose, Transpose, and Conjugate. Each routine adds the possibility of scaling during the transposition operation by giving some *alpha* and/or *beta* parameters. Each routine supports both row-major orderings and column-major orderings.

Table “BLAS-like Extensions” lists these routines.

The `<?>` symbol in the routine short names is a precision prefix that indicates the data type:

<i>s</i>	float
<i>d</i>	double
<i>c</i>	MKL_Complex8
<i>z</i>	MKL_Complex16

BLAS-like Extensions

Routine	Data Types	Description
<code>cblas_?axpby</code>	<i>s</i> , <i>d</i> , <i>c</i> , <i>z</i>	Scales two vectors, adds them to one another and stores result in the vector (routines).
<code>cblas_?axpy_batch</code> <code>cblas_?axpy_batch_strided</code>	<i>s</i> , <i>d</i> , <i>c</i> , <i>z</i>	Computes groups of vector-scalar products added to a vector.
<code>cblas_?dgmm_batch_strided</code> <code>cblas_?dgmm_batch</code>	<i>s</i> , <i>d</i> , <i>c</i> , <i>z</i>	Computes groups of diagonal matrix-general matrix product
<code>cblas_?gemm_batch</code> <code>cblas_?gemm_batch_strided</code>	<i>s</i> , <i>d</i> , <i>c</i> , <i>z</i>	Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.
<code>cblas_gemm_bf16bf16bf3bf</code>	bf16, bf16, bf3, bf	Computes a matrix-matrix product with general matrices of bf16 data type.

Routine	Data Types	Description
<code>cblas_gemm_bf16bf16f32_compute</code>	bfloat16	Computes a matrix-matrix product with general matrices of bfloat16 data type where one or both input matrices are stored in a packed data structure, and adds the result to a scalar-matrix product.
<code>cblas_gemm_f16f16f32_half_precision</code>	half precision	Computes a matrix-matrix product with general matrices of half precision data type.
<code>cblas_gemm_f16f16f32_half_precision_compute</code>	half precision	Computes a matrix-matrix product with general matrices of half precision data type where one or both input matrices are stored in a packed data structure, and adds the result to a scalar-matrix product.
<code>cblas_gemm_*</code>	Integer	Computes a matrix-matrix product with general integer matrices.
<code>cblas_?gemm_compute</code>	h, s, d	Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.
<code>cblas_gemm_*_compute</code>	Integer	Computes a matrix-matrix product with general integer matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.
<code>cblas_?gemm_pack</code>	h, s, d	Performs scaling and packing of the matrix into the previously allocated buffer.
<code>cblas_gemm_*_pack</code>	Integer, bfloat16	Pack the matrix into the buffer allocated previously.
<code>cblas_?gemm_pack_get_bytes</code>	h, s, d	Returns the number of bytes required to store the packed matrix.
<code>cblas_gemm_*_pack_get_bytes</code>	Integer, bfloat16	Returns the number of bytes required to store the packed matrix.
<code>cblas_?gemm3m</code>	c, z	Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.
<code>cblas_?gemm3m_batch</code>	c, z	Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.
<code>cblas_?gemm3m_batch_strided</code>		
<code>cblas_?gemmt</code>	s, d, c, z	Computes a matrix-matrix product with general matrices but updates only the upper or lower triangular part of the result matrix.
<code>cblas_?gemv_batch_strided</code>	s, d, c, z	Computes groups of matrix-vector product using general matrices.
<code>cblas_?gemv_batch</code>		
<code>cblas_?trsm_batch</code>	s, d, c, z	Solves a triangular matrix equation for a group of matrices.
<code>?cblas_?trsm_batch_strided</code>		
<code>mkl_?imatcopy</code>	s, d, c, z	Performs scaling and in-place transposition/copying of matrices.
<code>mkl_?imatcopy_batch_strided</code>	s, d, c, z	Computes groups of in-place matrix copy/transposition with scaling using general matrices.
<code>mkl_?imatcopy_batch</code>		

Routine	Data Types	Description
<code>mkl_?omatadd</code>	s, d, c, z	Performs scaling and sum of two matrices including their out-of-place transposition/copying.
<code>mkl_?omatcopy</code>	s, d, c, z	Performs scaling and out-of-place transposition/copying of matrices.
<code>mkl_? omatcopy_batch_stride d</code>	s, d, c, z	Computes groups of out of place matrix copy/transposition with scaling using general matrices.
<code>mkl_?omatcopy_batch</code>		
<code>mkl_?omatcopy2</code>	s, d, c, z	Performs two-strided scaling and out-of-place transposition/copying of matrices.
<code>mkl_jit_create_?gemm</code>	s, d, c, z	Creates a handle on a jitter and generates a GEMM kernel that computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product, with general matrices.
<code>mkl_jit_destroy</code>		Deletes the previously created jitter and the generated GEMM kernel.
<code>mkl_jit_get_?gemm_ptr</code>	s, d, c, z	Returns the GEMM kernel previously generated.

cblas_?axpy_batch

Computes a group of vector-scalar products added to a vector.

Syntax

```
void cblas_saxpy_batch (const MKL_INT *n_array, const float *alpha_array, const float
**x_array, const MKL_INT *incx_array, float **y_array, const MKL_INT *incy_array, const
MKL_INT group_count, const MKL_INT *group_size_array);
```

```
void cblas_daxpy_batch (const MKL_INT *n_array, const double *alpha_array, const double
**x_array, const MKL_INT *incx_array, double **y_array, const MKL_INT *incy_array,
const MKL_INT group_count, const MKL_INT *group_size_array);
```

```
void cblas_caxpy_batch (const MKL_INT *n_array, const void *alpha_array, const void
**x_array, const MKL_INT *incx_array, void **y_array, const MKL_INT *incy_array, const
MKL_INT group_count, const MKL_INT *group_size_array);
```

```
void cblas_zaxpy_batch (const MKL_INT *n_array, const void *alpha_array, const void
**x_array, const MKL_INT *incx_array, void **y_array, const MKL_INT *incy_array, const
MKL_INT group_count, const MKL_INT *group_size_array);
```

Description

The `cblas_?axpy_batch` routines perform a series of scalar-vector product added to a vector. They are similar to the `cblas_?axpy` routine counterparts, but the `cblas_?axpy_batch` routines perform vector operations with a group of vectors. The groups contain vectors with the same parameters.

The operation is defined as

```
idx = 0
for i = 0 ... group_count - 1
    n, alpha, incx, incy and group_size at position i in n_array, alpha_array, incx_array,
    incy_array and group_size_array
    for j = 0 ... group_size - 1
```



```

        x and y are vectors of size n at position idx in x_array and y_array
        y := alpha * x + y
        idx := idx + 1
    end for
end for

```

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

Input Parameters

<i>n_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , <i>n_i</i> = <i>n_array</i> [<i>i</i>] is the number of elements in vectors <i>x</i> and <i>y</i> .
<i>alpha_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , <i>alpha_i</i> = <i>alpha_array</i> [<i>i</i>] is the scalar <i>alpha</i> .
<i>x_array</i>	Array of size <i>total_batch_count</i> of pointers used to store <i>x</i> vectors. The array allocated for the <i>x</i> vectors of the group <i>i</i> must be of size at least $(1 + (n_i - 1) * \text{abs}(\text{incx}_i))$.
<i>incx_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , <i>incx_i</i> = <i>incx_array</i> [<i>i</i>] is the stride of vector <i>x</i> .
<i>y_array</i>	Array of size <i>total_batch_count</i> of pointers used to store <i>y</i> vectors. The array allocated for the <i>y</i> vectors of the group <i>i</i> must be of size at least $(1 + (n_i - 1) * \text{abs}(\text{incy}_i))$.
<i>incy_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , <i>incy_i</i> = <i>incy_array</i> [<i>i</i>] is the stride of vector <i>y</i> .
<i>group_count</i>	Number of groups. Must be at least 0.
<i>group_size_array</i>	Array of size <i>group_count</i> . The element <i>group_size_array</i> [<i>i</i>] is the number of vector in the group <i>i</i> . Each element in <i>group_size_array</i> must be at least 0.

Output Parameters

<i>y_array</i>	Array of pointers holding the <i>total_batch_count</i> updated vector <i>y</i> .
----------------	--

cblas_?axpy_batch_strided

Computes a group of vector-scalar products added to a vector.

Syntax

```
void cblas_saxpy_batch_strided (const MKL_INT n, const float alpha, const float *x,
const MKL_INT incx, const MKL_INT stridex, float *y, const MKL_INT incy, const MKL_INT
stridey, const MKL_INT batch_size);
```

```
void cblas_daxpy_batch_strided (const MKL_INT n, const double alpha, const double *x,
const MKL_INT incx, const MKL_INT stridex, double *y, const MKL_INT incy, const MKL_INT
stridey, const MKL_INT batch_size);
```

```
void cblas_caxpy_batch_strided (const MKL_INT n, const void alpha, const void *x, const
MKL_INT incx, const MKL_INT stridex, void *y, const MKL_INT incy, const MKL_INT
stridey, const MKL_INT batch_size);
```

```
void cblas_zaxpy_batch_strided (const MKL_INT n, const void alpha, const void *x, const
MKL_INT incx, const MKL_INT stridex, void *y, const MKL_INT incy, const MKL_INT
stridey, const MKL_INT batch_size);
```

Include Files

- mkl.h

Description

The `cblas_?axpy_batch_strided` routines perform a series of scalar-vector product added to a vector. They are similar to the `cblas_?axpy` routine counterparts, but the `cblas_?axpy_batch_strided` routines perform vector operations with a group of vectors.

All vector x (respectively, y) have the same parameters (size, increments) and are stored at constant *stridex* (respectively, *stridey*) from each other. The operation is defined as

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

Input Parameters

<i>n</i>	Number of elements in vectors x and y .
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>x</i>	Array of size at least <i>stridex</i> * <i>batch_size</i> holding the x vectors.
<i>incx</i>	Specifies the increment for the elements of x .
<i>stridex</i>	Stride between two consecutive x vectors; must be at least zero.
<i>y</i>	Array of size at least <i>stridey</i> * <i>batch_size</i> holding the y vectors.
<i>incy</i>	Specifies the increment for the elements of y .
<i>stridey</i>	Stride between two consecutive y vectors; must be at least $(1 + (n-1)*abs(incy))$.
<i>batch_size</i>	Number of <code>axpy</code> computations to perform and x and y vectors. Must be at least 0.

Output Parameters

<i>y</i>	Array holding the <i>batch_size</i> updated vector y .
----------	--

cblas_?axpyb

Scales two vectors, adds them to one another and stores result in the vector.

Syntax

```
void cblas_saxpyb (const MKL_INT n, const float a, const float *x, const MKL_INT incx,
const float b, float *y, const MKL_INT incy);
```

```
void cblas_daxpyb (const MKL_INT n, const double a, const double *x, const MKL_INT
incx, const double b, double *y, const MKL_INT incy);
```

```
void cblas_caxpby (const MKL_INT n, const void *a, const void *x, const MKL_INT incx,
const void *b, void *y, const MKL_INT incy);
```

```
void cblas_zaxpby (const MKL_INT n, const void *a, const void *x, const MKL_INT incx,
const void *b, void *y, const MKL_INT incy);
```

Include Files

- mkl.h

Description

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

$$y := a*x + b*y$$

where:

a and b are scalars

x and y are vectors each with n elements.

Input Parameters

n	Specifies the number of elements in vectors x and y .
a	Specifies the scalar a .
x	Array, size at least $(1 + (n-1)*abs(incx))$.
$incx$	Specifies the increment for the elements of x .
b	Specifies the scalar b .
y	Array, size at least $(1 + (n-1)*abs(incy))$.
$incy$	Specifies the increment for the elements of y .

Output Parameters

y	Contains the updated vector y .
-----	-----------------------------------

Example

For examples of routine usage, see these code examples in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- cblas_saxpby: examples\cblas\source\cblas_saxpbyx.c
- cblas_daxpby: examples\cblas\source\cblas_daxpbyx.c
- cblas_caxpby: examples\cblas\source\cblas_caxpbyx.c
- cblas_zaxpby: examples\cblas\source\cblas_zaxpbyx.c

cblas_?gemmt

Computes a matrix-matrix product with general matrices but updates only the upper or lower triangular part of the result matrix.

Syntax

```
void cblas_sgemmt (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT n, const MKL_INT k,
const float alpha, const float *a, const MKL_INT lda, const float *b, const MKL_INT
ldb, const float beta, float *c, const MKL_INT ldc);
```

```
void cblas_dgemmt (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT n, const MKL_INT k,
const double alpha, const double *a, const MKL_INT lda, const double *b, const MKL_INT
ldb, const double beta, double *c, const MKL_INT ldc);
```

```
void cblas_cgemmt (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT n, const MKL_INT k,
const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb,
const void *beta, void *c, const MKL_INT ldc);
```

```
void cblas_zgemmt (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT n, const MKL_INT k,
const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb,
const void *beta, void *c, const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The `?gemmt` routines compute a scalar-matrix-matrix product with general matrices and add the result to the upper or lower part of a scalar-matrix product. These routines are similar to the `?gemm` routines, but they only access and update a triangular part of the square result matrix (see Application Notes below).

The operation is defined as

$$C := \alpha \text{op}(A) * \text{op}(B) + \beta C,$$

where:

$\text{op}(X)$ is one of $\text{op}(X) = X$, or $\text{op}(X) = X^T$, or $\text{op}(X) = X^H$,

α and β are scalars,

A , B and C are matrices:

$\text{op}(A)$ is an n -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an n -by- n upper or lower triangular matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'CblasUpper', then the upper triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'CblasLower', then the lower triangular part of the array <i>c</i> is used.
<i>transa</i>	Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if <i>transa</i> = 'CblasNoTrans', then $\text{op}(A) = A$;

if *transa* = 'CblasTrans', then $\text{op}(A) = A^T$;
 if *transa* = 'CblasConjTrans', then $\text{op}(A) = A^H$.

transb

Specifies the form of $\text{op}(B)$ used in the matrix multiplication:

if *transb* = 'CblasNoTrans', then $\text{op}(B) = B$;
 if *transb* = 'CblasTrans', then $\text{op}(B) = B^T$;
 if *transb* = 'CblasConjTrans', then $\text{op}(B) = B^H$.

n

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

k

Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of *k* must be at least zero.

alpha

Specifies the scalar *alpha*.

a

	<i>transa</i> ='CblasNoTrans'	<i>transa</i> ='CblasTrans' or 'CblasConjTrans'
<i>Layout</i> ='CblasColMajor' or ' <i>Layout</i> ='CblasRowMajor' or '	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size <i>lda</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
	Array, size <i>lda</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size <i>lda</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .

lda

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

	<i>transa</i> ='CblasNoTrans'	<i>transa</i> ='CblasTrans' or 'CblasConjTrans'
<i>Layout</i> ='CblasColMajor' or ' <i>Layout</i> ='CblasRowMajor' or '	<i>lda</i> must be at least $\max(1, n)$.	<i>lda</i> must be at least $\max(1, k)$.
	<i>lda</i> must be at least $\max(1, k)$.	<i>lda</i> must be at least $\max(1, n)$.

b

	<i>transb</i> ='CblasNoTrans'	<i>transb</i> ='CblasTrans' or 'CblasConjTrans'
<i>Layout</i> ='CblasColMajor' or ' <i>Layout</i> ='CblasRowMajor' or '	Array, size <i>ldb</i> * <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>ldb</i> * <i>k</i> . Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .

<i>Layout</i> = 'CblasRowMaj or'	Array, size $ldb * k$. Before entry, the leading n -by- k part of the array b must contain the matrix B .	Array, size $ldb * n$. Before entry, the leading k -by- n part of the array b must contain the matrix B .
-------------------------------------	--	--

ldb

Specifies the leading dimension of b as declared in the calling (sub)program.

	<i>transb</i> = 'CblasNoTr ans'	<i>transb</i> = 'CblasTran s' or 'CblasConjTrans'
<i>Layout</i> = 'CblasColMaj or'	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> must be at least $\max(1, n)$.
<i>Layout</i> = 'CblasRowMaj or'	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$.

beta

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

c

Array, size ldc by n .

When *beta* is equal to zero, *c* need not be set on input.

<i>uplo</i> = 'CblasUpper'	<i>uplo</i> = 'CblasLower'
The leading n -by- n upper triangular part of the array <i>c</i> must contain the upper triangular part of the matrix C and the strictly lower triangular part of <i>c</i> is not referenced.	The leading n -by- n lower triangular part of the array <i>c</i> must contain the lower triangular part of the matrix C and the strictly upper triangular part of <i>c</i> is not referenced.

ldc

Specifies the leading dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least $\max(1, n)$.

Output Parameters

c

When *uplo* = 'CblasUpper', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

When *uplo* = 'CblasLower', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

Application Notes

These routines only access and update the upper or lower triangular part of the result matrix. This can be useful when the result is known to be symmetric; for example, when computing a product of the form $C := \alpha * B * S * B^T + \beta * C$, where S and C are symmetric matrices and B is a general matrix. In this case, first compute $A := B * S$ (which can be done using the corresponding ?*symm* routine), then compute $C := \alpha * A * B^T + \beta * C$ using the ?*gemmt* routine.

cblas_?gemm3m

Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.

Syntax

```
void cblas_cgemm3m (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void
*alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void
*beta, void *c, const MKL_INT ldc);
```

```
void cblas_zgemm3m (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void
*alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void
*beta, void *c, const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The ?gemm3m routines perform a matrix-matrix operation with general complex matrices. These routines are similar to the ?gemm routines, but they use fewer matrix multiplication operations (see *Application Notes* below).

The operation is defined as

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C,$$

where:

$\text{op}(x)$ is one of $\text{op}(x) = x$, or $\text{op}(x) = x'$, or $\text{op}(x) = \text{conjg}(x')$,

α and β are scalars,

A , B and C are matrices:

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>transa</i>	Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if $\text{transa}=\text{CblasNoTrans}$, then $\text{op}(A) = A$; if $\text{transa}=\text{CblasTrans}$, then $\text{op}(A) = A'$; if $\text{transa}=\text{CblasConjTrans}$, then $\text{op}(A) = \text{conjg}(A')$.
<i>transb</i>	Specifies the form of $\text{op}(B)$ used in the matrix multiplication: if $\text{transb}=\text{CblasNoTrans}$, then $\text{op}(B) = B$; if $\text{transb}=\text{CblasTrans}$, then $\text{op}(B) = B'$; if $\text{transb}=\text{CblasConjTrans}$, then $\text{op}(B) = \text{conjg}(B')$.
<i>m</i>	Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C . The value of m must be at least zero.

n Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C .

The value of n must be at least zero.

k Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$.

The value of k must be at least zero.

α Specifies the scalar α .

a

	$\text{transa}=\text{CblasNoTrans}$	$\text{transa}=\text{CblasTrans}$ or $\text{transa}=\text{CblasConjTrans}$
$\text{Layout} = \text{CblasColMajor}$	Array, size $\text{lda} * k$. Before entry, the leading m -by- k part of the array a must contain the matrix A .	Array, size $\text{lda} * m$. Before entry, the leading k -by- m part of the array a must contain the matrix A .
$\text{Layout} = \text{CblasRowMajor}$	Array, size $\text{lda} * m$. Before entry, the leading k -by- m part of the array a must contain the matrix A .	Array, size $\text{lda} * k$. Before entry, the leading m -by- k part of the array a must contain the matrix A .

lda

Specifies the leading dimension of a as declared in the calling (sub)program.

	$\text{transa}=\text{CblasNoTrans}$	$\text{transa}=\text{CblasTrans}$ or $\text{transa}=\text{CblasConjTrans}$
$\text{Layout} = \text{CblasColMajor}$	lda must be at least $\max(1, m)$.	lda must be at least $\max(1, k)$
$\text{Layout} = \text{CblasRowMajor}$	lda must be at least $\max(1, k)$	lda must be at least $\max(1, m)$.

b

	$\text{transb}=\text{CblasNoTrans}$	$\text{transb}=\text{CblasTrans}$ or $\text{transb}=\text{CblasConjTrans}$
$\text{Layout} = \text{CblasColMajor}$	Array, size ldb by n . Before entry, the leading k -by- n part of the array b must contain the matrix B .	Array, size ldb by k . Before entry the leading n -by- k part of the array b must contain the matrix B .

<i>Layout</i> = CblasRowMajor	Array, size <i>ldb</i> by <i>k</i> . Before entry the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>ldb</i> by <i>n</i> . Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
----------------------------------	---	---

ldb

Specifies the leading dimension of *b* as declared in the calling (sub)program.

	<i>transb</i> =CblasNoTrans	<i>transb</i> =CblasTrans or <i>transb</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> must be at least $\max(1, n)$.
<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$.

beta

Specifies the scalar *beta*.

When *beta* is equal to zero, then *c* need not be set on input.

c

<i>Layout</i> = CblasColMajor	Array, size <i>ldc</i> by <i>n</i> . Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.
<i>Layout</i> = CblasRowMajor	Array, size <i>ldc</i> by <i>m</i> . Before entry, the leading <i>n</i> -by- <i>m</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.

ldc

Specifies the leading dimension of *c* as declared in the calling (sub)program.

<i>Layout</i> = CblasColMajor	<i>ldc</i> must be at least $\max(1, m)$.
<i>Layout</i> = CblasRowMajor	<i>ldc</i> must be at least $\max(1, n)$.

Output Parameters

c

Overwritten by the *m*-by-*n* matrix $(\alpha * \text{op}(A) * \text{op}(B) + \beta * C)$.

Application Notes

These routines perform a complex matrix multiplication by forming the real and imaginary parts of the input matrices. This uses three real matrix multiplications and five real matrix additions instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications reduces the time spent in matrix operations by 25%, resulting in significant savings in compute time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

$fl(x \text{ op } y) = (x \text{ op } y)(1 + \delta)$, $|\delta| \leq u$, $\text{op} = \times, /$, $fl(x \pm y) = x(1 + \alpha) \pm y(1 + \beta)$, $|\alpha|, |\beta| \leq u$

then for an n -by- n matrix $\hat{C} = fl(C_1 + iC_2) = fl((A_1 + iA_2)(B_1 + iB_2)) = \hat{C}_1 + i\hat{C}_2$, the following bounds are satisfied:

$$\|\hat{C}_1 - C_1\| \leq 2(n+1)u\|A\|_\infty\|B\|_\infty + O(u^2),$$

$$\|\hat{C}_2 - C_2\| \leq 4(n+4)u\|A\|_\infty\|B\|_\infty + O(u^2),$$

where $\|A\|_\infty = \max(\|A_1\|_\infty, \|A_2\|_\infty)$, and $\|B\|_\infty = \max(\|B_1\|_\infty, \|B_2\|_\infty)$.

Thus the corresponding matrix multiplications are stable.

cblas_?gemm_batch

Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.

Syntax

```
void cblas_sgemm_batch (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE* transa_array,
const CBLAS_TRANSPOSE* transb_array, const MKL_INT* m_array, const MKL_INT* n_array,
const MKL_INT* k_array, const float* alpha_array, const float **a_array, const MKL_INT*
lda_array, const float **b_array, const MKL_INT* ldb_array, const float* beta_array,
float **c_array, const MKL_INT* ldc_array, const MKL_INT group_count, const MKL_INT*
group_size);
```

```
void cblas_dgemm_batch (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE* transa_array,
const CBLAS_TRANSPOSE* transb_array, const MKL_INT* m_array, const MKL_INT* n_array,
const MKL_INT* k_array, const double* alpha_array, const double **a_array, const
MKL_INT* lda_array, const double **b_array, const MKL_INT* ldb_array, const double*
beta_array, double **c_array, const MKL_INT* ldc_array, const MKL_INT group_count,
const MKL_INT* group_size);
```

```
void cblas_cgemm_batch (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE* transa_array,
const CBLAS_TRANSPOSE* transb_array, const MKL_INT* m_array, const MKL_INT* n_array,
const MKL_INT* k_array, const void *alpha_array, const void **a_array, const MKL_INT*
lda_array, const void **b_array, const MKL_INT* ldb_array, const void *beta_array, void
**c_array, const MKL_INT* ldc_array, const MKL_INT group_count, const MKL_INT*
group_size);
```

```
void cblas_zgemm_batch (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE* transa_array,
const CBLAS_TRANSPOSE* transb_array, const MKL_INT* m_array, const MKL_INT* n_array,
const MKL_INT* k_array, const void *alpha_array, const void **a_array, const MKL_INT*
lda_array, const void **b_array, const MKL_INT* ldb_array, const void *beta_array, void
**c_array, const MKL_INT* ldc_array, const MKL_INT group_count, const MKL_INT*
group_size);
```

Include Files

- mkl.h

Description

The ?gemm_batch routines perform a series of matrix-matrix operations with general matrices. They are similar to the ?gemm routine counterparts, but the ?gemm_batch routines perform matrix-matrix operations with groups of matrices, processing a number of groups at once. The groups contain matrices with the same parameters.

The operation is defined as

```

idx = 0
for i = 0..group_count - 1
  alpha and beta in alpha_array[i] and beta_array[i]
  for j = 0..group_size[i] - 1
    A, B, and C matrix in a_array[idx], b_array[idx], and c_array[idx]
    C := alpha*op(A)*op(B) + beta*C,
    idx = idx + 1
  end for
end for

```

where:

$\text{op}(X)$ is one of $\text{op}(X) = X$, or $\text{op}(X) = X^T$, or $\text{op}(X) = X^H$,

alpha and *beta* are scalar elements of *alpha_array* and *beta_array*,

A, *B* and *C* are matrices such that for *m*, *n*, and *k* which are elements of *m_array*, *n_array*, and *k_array*:

$\text{op}(A)$ is an *m*-by-*k* matrix,

$\text{op}(B)$ is a *k*-by-*n* matrix,

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

A, *B*, and *C* represent matrices stored at addresses pointed to by *a_array*, *b_array*, and *c_array*, respectively. The number of entries in *a_array*, *b_array*, and *c_array* is *total_batch_count* = the sum of all of the *group_size* entries.

See also [gemm](#) for a detailed description of multiplication for general matrices and [?gemm3m_batch](#), BLAS-like extension routines for similar matrix-matrix operations.

NOTE

Error checking is not performed for oneMKL Windows* single dynamic libraries for the `?gemm_batch` routines.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>transa_array</i>	<p>Array of size <i>group_count</i>. For the group <i>i</i>, <i>transa_i</i> = <i>transa_array</i>[<i>i</i>] specifies the form of $\text{op}(A)$ used in the matrix multiplication:</p> <p>if <i>transa_i</i> = CblasNoTrans, then $\text{op}(A) = A$;</p> <p>if <i>transa_i</i> = CblasTrans, then $\text{op}(A) = A^T$;</p> <p>if <i>transa_i</i> = CblasConjTrans, then $\text{op}(A) = A^H$.</p>
<i>transb_array</i>	<p>Array of size <i>group_count</i>. For the group <i>i</i>, <i>transb_i</i> = <i>transb_array</i>[<i>i</i>] specifies the form of $\text{op}(B)$ used in the matrix multiplication:</p> <p>if <i>transb_i</i> = CblasNoTrans, then $\text{op}(B) = B$;</p> <p>if <i>transb_i</i> = CblasTrans, then $\text{op}(B) = B^T$;</p> <p>if <i>transb_i</i> = CblasConjTrans, then $\text{op}(B) = B^H$.</p>
<i>m_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , <i>m_i</i> = <i>m_array</i> [<i>i</i>] specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix <i>C</i> .

The value of each element of m_array must be at least zero.

n_array

Array of size $group_count$. For the group i , $n_i = n_array[i]$ specifies the number of columns of the matrix $op(B)$ and the number of columns of the matrix C .

The value of each element of n_array must be at least zero.

k_array

Array of size $group_count$. For the group i , $k_i = k_array[i]$ specifies the number of columns of the matrix $op(A)$ and the number of rows of the matrix $op(B)$.

The value of each element of k_array must be at least zero.

$alpha_array$

Array of size $group_count$. For the group i , $alpha_array[i]$ specifies the scalar $alpha_i$.

a_array

Array, size $total_batch_count$, of pointers to arrays used to store A matrices.

lda_array

Array of size $group_count$. For the group i , $lda_i = lda_array[i]$ specifies the leading dimension of the array storing matrix A as declared in the calling (sub)program.

	$transa_i = \text{CblasNoTrans}$	$transa_i = \text{CblasTrans}$ or $transa_i = \text{CblasConjTrans}$
$Layout = \text{CblasColMajor}$	lda_i must be at least $\max(1, m_i)$.	lda_i must be at least $\max(1, k_i)$
$Layout = \text{CblasRowMajor}$	lda_i must be at least $\max(1, k_i)$	lda_i must be at least $\max(1, m_i)$.

b_array

Array, size $total_batch_count$, of pointers to arrays used to store B matrices.

ldb_array

Array of size $group_count$. For the group i , $ldb_i = ldb_array[i]$ specifies the leading dimension of the array storing matrix B as declared in the calling (sub)program.

	$transb_i = \text{CblasNoTrans}$	$transb_i = \text{CblasTrans}$ or $transb_i = \text{CblasConjTrans}$
$Layout = \text{CblasColMajor}$	ldb_i must be at least $\max(1, k_i)$.	ldb_i must be at least $\max(1, n_i)$.
$Layout = \text{CblasRowMajor}$	ldb_i must be at least $\max(1, n_i)$.	ldb_i must be at least $\max(1, k_i)$.

$beta_array$

Array of size $group_count$. For the group i , $beta_array[i]$ specifies the scalar $beta_i$.

When $beta_i$ is equal to zero, then C matrices in group i need not be set on input.

<code>c_array</code>	Array, size <code>total_batch_count</code> , of pointers to arrays used to store <i>C</i> matrices.
<code>ldc_array</code>	<p>Array of size <code>group_count</code>. For the group <i>i</i>, <code>ldc_i = ldc_array[i]</code> specifies the leading dimension of all arrays storing matrix <i>C</i> in group <i>i</i> as declared in the calling (sub)program.</p> <p>When <code>Layout = CblasColMajor</code>/<code>ldc_i</code> must be at least $\max(1, m_i)$.</p> <p>When <code>Layout = CblasRowMajor</code>/<code>ldc_i</code> must be at least $\max(1, n_i)$.</p>
<code>group_count</code>	Specifies the number of groups. Must be at least 0.
<code>group_size</code>	Array of size <code>group_count</code> . The element <code>group_size[i]</code> specifies the number of matrices in group <i>i</i> . Each element in <code>group_size</code> must be at least 0.

Output Parameters

<code>c_array</code>	Output buffer, overwritten by <code>total_batch_count</code> matrix multiply operations of the form $\alpha * \text{op}(A) * \text{op}(B) + \beta * C$.
----------------------	--

cblas_?gemm_batch_strided

Computes groups of matrix-matrix product with general matrices.

Syntax

```
void cblas_sgemm_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE
transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT
k, const float alpha, const float *a, const MKL_INT lda, const MKL_INT stridea, const
float *b, const MKL_INT ldb, const MKL_INT strideb, const float beta, float *c, const
MKL_INT ldc, const MKL_INT stridec, const MKL_INT batch_size);
```

```
void cblas_dgemm_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE
transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT
k, const double alpha, const double *a, const MKL_INT lda, const MKL_INT stridea, const
double *b, const MKL_INT ldb, const MKL_INT strideb, const double beta, double *c,
const MKL_INT ldc, const MKL_INT stridec, const MKL_INT batch_size);
```

```
void cblas_cgemm_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE
transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT
k, const void *alpha, const void *a, const MKL_INT lda, const MKL_INT stridea, const
void *b, const MKL_INT ldb, const MKL_INT strideb, const void *beta, void *c, const
MKL_INT ldc, const MKL_INT stridec, const MKL_INT batch_size);
```

```
void cblas_zgemm_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE
transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT
k, const void *alpha, const void *a, const MKL_INT lda, const MKL_INT stridea, const
void *b, const MKL_INT ldb, const MKL_INT strideb, const void *beta, void *c, const
MKL_INT ldc, const MKL_INT stridec, const MKL_INT batch_size);
```

Include Files

- `mk1.h`

Description

The `cblas_?gemm_batch_strided` routines perform a series of matrix-matrix operations with general matrices. They are similar to the `cblas_?gemm` routine counterparts, but the `cblas_?gemm_batch_strided` routines perform matrix-matrix operations with groups of matrices. The groups contain matrices with the same parameters.

All matrix *a* (respectively, *b* or *c*) have the same parameters (size, leading dimension, transpose operation, alpha, beta scaling) and are stored at constant *stridea* (respectively, *strideb* or *stridedc*) from each other. The operation is defined as

```
For i = 0 ... batch_size - 1
    Ai, Bi and Ci are matrices at offset i * stridea, i * strideb and i * stridedc in a, b and c
    Ci = alpha * Ai * Bi + beta * Ci
end for
```

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>transa</i>	Specifies $\text{op}(A)$ the transposition operation applied to the matrices <i>A</i> . if <i>transa</i> = CblasNoTrans, then $\text{op}(A) = A$; if <i>transa</i> = CblasTrans, then $\text{op}(A) = A^T$; if <i>transa</i> = CblasConjTrans, then $\text{op}(A) = A^H$.
<i>transb</i>	Specifies $\text{op}(B)$ the transposition operation applied to the matrices <i>B</i> . if <i>transb</i> = CblasNoTrans, then $\text{op}(B) = B$; if <i>transb</i> = CblasTrans, then $\text{op}(B) = B^T$; if <i>transb</i> = CblasConjTrans, then $\text{op}(B) = B^H$.
<i>m</i>	Number of rows of the $\text{op}(A)$ and <i>C</i> matrices. Must be at least 0.
<i>n</i>	Number of columns of the $\text{op}(B)$ and <i>C</i> matrices. Must be at least 0.
<i>k</i>	Number of columns of the $\text{op}(A)$ matrix and number of rows of the $\text{op}(B)$ matrix. Must be at least 0.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>a</i>	Array of size at least <i>stridea</i> * <i>batch_size</i> holding the <i>a</i> matrices.

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans or CblasConjTrans
<i>layout</i> = CblasColMajor	Before entry, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> + <i>i</i> * <i>stridea</i> must contain the matrix <i>A_i</i> .	Before entry, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> + <i>i</i> * <i>stridea</i> must contain the matrix <i>A_i</i> .
<i>layout</i> = CblasRowMajor	Before entry, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> + <i>i</i> * <i>stridea</i> must contain the matrix <i>A_i</i> .	Before entry, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> + <i>i</i> * <i>stridea</i> must contain the matrix <i>A_i</i> .

*lda*Specifies the leading dimension of the *a* matrices.

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans or CblasConjTrans
<i>layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1,m)$	<i>lda</i> must be at least $\max(1,k)$.
<i>layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1,k)$.	<i>lda</i> must be at least $\max(1,m)$

*stridea*Stride between two consecutive *a* matrices.

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans or CblasConjTrans
<i>layout</i> = CblasColMajor	Must be at least $lda*k$	Must be at least $lda*m$
<i>layout</i> = CblasRowMajor	Must be at least $lda*m$	Must be at least $lda*k$

*b*Array of size at least *strideb***batch_size* holding the *b* matrices.

	<i>transb</i> =CblasNoTrans	<i>transb</i> =CblasTrans or CblasConjTrans
<i>layout</i> = CblasColMajor	Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> + <i>i</i> * <i>strideb</i> must contain the matrix <i>B_i</i> .	Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> + <i>i</i> * <i>strideb</i> must contain the matrix <i>B_i</i> .
<i>layout</i> = CblasRowMajor	Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> + <i>i</i> * <i>strideb</i> must contain the matrix <i>B_i</i> .	Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> + <i>i</i> * <i>strideb</i> must contain the matrix <i>B_i</i> .

*ldb*Specifies the leading dimension of the *b* matrices.

	<i>transab</i> =CblasNoTrans	<i>transb</i> =CblasTrans or CblasConjTrans
<i>layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1,k)$	<i>ldb</i> must be at least $\max(1,n)$.
<i>layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1,n)$.	<i>ldb</i> must be at least $\max(1,k)$

*strideb*Stride between two consecutive *b* matrices.

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans or CblasConjTrans
--	-----------------------------	---

<code>layout = CblasColMajor</code>	Must be at least $ldb*n$	Must be at least $ldb*k$
<code>layout = CblasRowMajor</code>	Must be at least $ldb*k$	Must be at least $ldb*n$

<code>beta</code>	Specifies the scalar <i>beta</i> .
<code>c</code>	<p>Array of size at least $stridec*batch_size$ holding the <i>c</i> matrices.</p> <p>If <code>layout=CblasColMajor</code>, before entry, the leading <i>m</i>-by-<i>n</i> part of the array $c + i * stridec$ must contain the matrix C_i.</p> <p>If <code>layout=CblasRowMajor</code>, before entry, the leading <i>n</i>-by-<i>m</i> part of the array $c + i * stridec$ must contain the matrix C_i.</p>
<code>ldc</code>	<p>Specifies the leading dimension of the <i>c</i> matrices.</p> <p>Must be at least $\max(1,m)$ if <code>layout=CblasColMajor</code> or $\max(1,n)$ if <code>layout=CblasRowMajor</code>.</p>
<code>stridec</code>	<p>Specifies the stride between two consecutive <i>c</i> matrices.</p> <p>Must be at least $ldc*n$ if <code>layout=CblasColMajor</code> or $ldc*m$ if <code>layout=CblasRowMajor</code>.</p>
<code>batch_size</code>	Number of <code>gemm</code> computations to perform and <i>a</i> , <i>b</i> and <i>c</i> matrices. Must be at least 0.

Output Parameters

<code>c</code>	Array holding the <i>batch_size</i> updated <i>c</i> matrices.
----------------	--

`cblas_?gemm3m_batch_strided`

Computes groups of matrix-matrix product with general matrices.

Syntax

```
void cblas_cgemm3m_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE
transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT
k, const void *alpha, const void *a, const MKL_INT lda, const MKL_INT stridea, const
void *b, const MKL_INT ldb, const MKL_INT strideb, const void *beta, void *c, const
MKL_INT ldc, const MKL_INT stridec, const MKL_INT batch_size);
```

```
void cblas_zgemm3m_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE
transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT
k, const void *alpha, const void *a, const MKL_INT lda, const MKL_INT stridea, const
void *b, const MKL_INT ldb, const MKL_INT strideb, const void *beta, void *c, const
MKL_INT ldc, const MKL_INT stridec, const MKL_INT batch_size);
```

Include Files

- `mkl.h`

Description

The `cblas_?gemm3m_batch_strided` routines perform a series of matrix-matrix operations with general matrices. They are similar to the `cblas_?gemm` routine counterparts, but the `cblas_?gemm3m_batch_strided` routines perform matrix-matrix operations with groups of matrices. The groups contain matrices with the same parameters.

All matrix a (respectively, b or c) have the same parameters (size, leading dimension, transpose operation, alpha, beta scaling) and are stored at constant *stridea* (respectively, *strideb* or *stridedc*) from each other. The operation is defined as

```
For i = 0 ... batch_size - 1
    Ai, Bi and Ci are matrices at offset i * stridea, i * strideb and i * stridedc in a, b and c
    Ci = alpha * Ai * Bi + beta * Ci
end for
```

The `cblas_?gemm3m_batch_strided` routines use fewer matrix multiplications than the `cblas_?gemm` routines, as described in the *Application Notes* below.

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>transa</i>	Specifies $\text{op}(A)$ the transposition operation applied to the matrices A . if <i>transa</i> = CblasNoTrans, then $\text{op}(A) = A$; if <i>transa</i> = CblasTrans, then $\text{op}(A) = A^T$; if <i>transa</i> = CblasConjTrans, then $\text{op}(A) = A^H$.
<i>transb</i>	Specifies $\text{op}(B)$ the transposition operation applied to the matrices B . if <i>transb</i> = CblasNoTrans, then $\text{op}(B) = B$; if <i>transb</i> = CblasTrans, then $\text{op}(B) = B^T$; if <i>transb</i> = CblasConjTrans, then $\text{op}(B) = B^H$.
<i>m</i>	Number of rows of the $\text{op}(A)$ and C matrices. Must be at least 0.
<i>n</i>	Number of columns of the $\text{op}(B)$ and C matrices. Must be at least 0.
<i>k</i>	Number of columns of the $\text{op}(A)$ matrix and number of rows of the $\text{op}(B)$ matrix. Must be at least 0.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>a</i>	Array of size at least <i>stridea</i> * <i>batch_size</i> holding the a matrices.

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans or CblasConjTrans
<i>layout</i> = CblasColMajor	Before entry, the leading m -by- k part of the array $a + i * \text{stridea}$ must contain the matrix A_i .	Before entry, the leading k -by- m part of the array $a + i * \text{stridea}$ must contain the matrix A_i .

<code>layout = CblasRowMajor</code>	Before entry, the leading k -by- m part of the array $a + i * stride_a$ must contain the matrix A_i .	Before entry, the leading m -by- k part of the array $a + i * stride_a$ must contain the matrix A_i .
-------------------------------------	---	---

*lda*Specifies the leading dimension of the a matrices.

	<code>transa=CblasNoTrans</code>	<code>transa=CblasTrans</code> or <code>CblasConjTrans</code>
<code>layout = CblasColMajor</code>	<i>lda</i> must be at least $\max(1, m)$.	<i>lda</i> must be at least $\max(1, k)$.
<code>layout = CblasRowMajor</code>	<i>lda</i> must be at least $\max(1, k)$.	<i>lda</i> must be at least $\max(1, m)$.

*stridea*Stride between two consecutive a matrices.

	<code>transa=CblasNoTrans</code>	<code>transa=CblasTrans</code> or <code>CblasConjTrans</code>
<code>layout = CblasColMajor</code>	Must be at least $lda * k$.	Must be at least $lda * m$.
<code>layout = CblasRowMajor</code>	Must be at least $lda * m$.	Must be at least $lda * k$.

*b*Array of size at least $stride_b * batch_size$ holding the b matrices.

	<code>transb=CblasNoTrans</code>	<code>transb=CblasTrans</code> or <code>CblasConjTrans</code>
<code>layout = CblasColMajor</code>	Before entry, the leading k -by- n part of the array $b + i * stride_b$ must contain the matrix B_i .	Before entry, the leading n -by- k part of the array $b + i * stride_b$ must contain the matrix B_i .
<code>layout = CblasRowMajor</code>	Before entry, the leading n -by- k part of the array $b + i * stride_b$ must contain the matrix B_i .	Before entry, the leading k -by- n part of the array $b + i * stride_b$ must contain the matrix B_i .

*ldb*Specifies the leading dimension of the b matrices.

	<code>transab=CblasNoTrans</code>	<code>transb=CblasTrans</code> or <code>CblasConjTrans</code>
<code>layout = CblasColMajor</code>	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> must be at least $\max(1, n)$.

<i>layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1,n)$.	<i>ldb</i> must be at least $\max(1,k)$.
----------------------------------	--	--

*strideb*Stride between two consecutive *b* matrices.

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans or CblasConjTrans
<i>layout</i> = CblasColMajor	Must be at least <i>ldb</i> * <i>n</i> .	Must be at least <i>ldb</i> * <i>k</i> .
<i>layout</i> = CblasRowMajor	Must be at least <i>ldb</i> * <i>k</i> .	Must be at least <i>ldb</i> * <i>n</i> .

*beta*Specifies the scalar *beta*.*c*Array of size at least *stridec***batch_size* holding the *c* matrices.

If *layout*=CblasColMajor, before entry, the leading *m*-by-*n* part of the array *c* + *i* * *stridec* must contain the matrix *C_i*.

If *layout*=CblasRowMajor, before entry, the leading *n*-by-*m* part of the array *c* + *i* * *stridec* must contain the matrix *C_i*.

*ldc*Specifies the leading dimension of the *c* matrices.

Must be at least $\max(1,m)$ if *layout*=CblasColMajor or $\max(1,n)$ if *layout*=CblasRowMajor.

*stridec*Specifies the stride between two consecutive *c* matrices.

Must be at least *ldc***n* if *layout*=CblasColMajor or *ldc***m* if *layout*=CblasRowMajor.

batch_size

Number of `gemm` computations to perform and *a*, *b* and *c* matrices. Must be at least 0.

Output Parameters

*c*Array holding the *batch_size* updated *c* matrices.

Application Notes

These routines perform a complex matrix multiplication by forming the real and imaginary parts of the input matrices. This uses three real matrix multiplications and five real matrix additions instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications reduces the time spent in matrix operations by 25%, resulting in significant savings in compute time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

$$fl(x \text{ op } y) = (x \text{ op } y) (1 + \delta), |\delta| \leq u, \text{ op} = *, /, fl(x \pm y) = x(1 + \alpha) \pm y(1 + \beta), |\alpha|, |\beta| \leq u$$

then for an *n*-by-*n* matrix $\hat{C} = fl(C1 + iC2) = fl((A1 + iA2)(B1 + iB2)) = \hat{C}1 + i\hat{C}2$, the following bounds are satisfied:

$$\begin{aligned} \|\hat{C}1 - C1\| &\leq 2(n+1)u \|A\|_{\infty} \|B\|_{\infty} + O(u^2), \\ \|\hat{C}2 - C2\| &\leq 4(n+4)u \|A\|_{\infty} \|B\|_{\infty} + O(u^2), \end{aligned}$$

where $\|A\|_{\infty} = \max(\|A1\|_{\infty}, \|A2\|_{\infty})$, and $\|B\|_{\infty} = \max(\|B1\|_{\infty}, \|B2\|_{\infty})$.

Thus the corresponding matrix multiplications are stable.

cblas_?gemm3m_batch

Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.

Syntax

```
void cblas_cgemm3m_batch (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE*
transa_array, const CBLAS_TRANSPOSE* transb_array, const MKL_INT* m_array, const
MKL_INT* n_array, const MKL_INT* k_array, const void *alpha_array, const void
**a_array, const MKL_INT* lda_array, const void **b_array, const MKL_INT* ldb_array,
const void *beta_array, void **c_array, const MKL_INT* ldc_array, const MKL_INT
group_count, const MKL_INT* group_size);
```

```
void cblas_zgemm3m_batch (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE*
transa_array, const CBLAS_TRANSPOSE* transb_array, const MKL_INT* m_array, const
MKL_INT* n_array, const MKL_INT* k_array, const void *alpha_array, const void
**a_array, const MKL_INT* lda_array, const void **b_array, const MKL_INT* ldb_array,
const void *beta_array, void **c_array, const MKL_INT* ldc_array, const MKL_INT
group_count, const MKL_INT* group_size);
```

Include Files

- mkl.h

Description

The ?gemm3m_batch routines perform a series of matrix-matrix operations with general matrices. They are similar to the ?gemm3m routine counterparts, but the ?gemm3m_batch routines perform matrix-matrix operations with groups of matrices, processing a number of groups at once. The groups contain matrices with the same parameters. The ?gemm3m_batch routines use fewer matrix multiplications than the ?gemm_batch routines, as described in the *Application Notes*.

The operation is defined as

```
idx = 0
for i = 0..group_count - 1
  alpha and beta in alpha_array[i] and beta_array[i]
  for j = 0..group_size[i] - 1
    A, B, and C matrix in a_array[idx], b_array[idx], and c_array[idx]
    C := alpha*op(A)*op(B) + beta*C,
    idx = idx + 1
  end for
end for
```

where:

$\text{op}(X)$ is one of $\text{op}(X) = X$, or $\text{op}(X) = X^T$, or $\text{op}(X) = X^H$,

α and β are scalar elements of α_array and β_array ,

A , B and C are matrices such that for m , n , and k which are elements of m_array , n_array , and k_array :

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

A , B , and C represent matrices stored at addresses pointed to by `a_array`, `b_array`, and `c_array`, respectively. The number of entries in `a_array`, `b_array`, and `c_array` is `total_batch_count` = the sum of all the `group_size` entries.

See also [gemm](#) for a detailed description of multiplication for general matrices and [gemm_batch](#), BLAS-like extension routines for similar matrix-matrix operations.

NOTE

Error checking is not performed for Intel® oneAPI Math Kernel Library (oneMKL) Windows* single dynamic libraries for the `?gemm3m_batch` routines.

Input Parameters

<code>Layout</code>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<code>transa_array</code>	<p>Array of size <code>group_count</code>. For the group i, $transa_i = transa_array[i]$ specifies the form of $op(A)$ used in the matrix multiplication:</p> <p>if $transa_i = \text{CblasNoTrans}$, then $op(A) = A$;</p> <p>if $transa_i = \text{CblasTrans}$, then $op(A) = A^T$;</p> <p>if $transa_i = \text{CblasConjTrans}$, then $op(A) = A^H$.</p>
<code>transb_array</code>	<p>Array of size <code>group_count</code>. For the group i, $transb_i = transb_array[i]$ specifies the form of $op(B_i)$ used in the matrix multiplication:</p> <p>if $transb_i = \text{CblasNoTrans}$, then $op(B) = B$;</p> <p>if $transb_i = \text{CblasTrans}$, then $op(B) = B^T$;</p> <p>if $transb_i = \text{CblasConjTrans}$, then $op(B) = B^H$.</p>
<code>m_array</code>	<p>Array of size <code>group_count</code>. For the group i, $m_i = m_array[i]$ specifies the number of rows of the matrix $op(A)$ and of the matrix C.</p> <p>The value of each element of <code>m_array</code> must be at least zero.</p>
<code>n_array</code>	<p>Array of size <code>group_count</code>. For the group i, $n_i = n_array[i]$ specifies the number of columns of the matrix $op(B)$ and the number of columns of the matrix C.</p> <p>The value of each element of <code>n_array</code> must be at least zero.</p>
<code>k_array</code>	<p>Array of size <code>group_count</code>. For the group i, $k_i = k_array[i]$ specifies the number of columns of the matrix $op(A)$ and the number of rows of the matrix $op(B)$.</p> <p>The value of each element of <code>k_array</code> must be at least zero.</p>
<code>alpha_array</code>	Array of size <code>group_count</code> . For the group i , <code>alpha_array[i]</code> specifies the scalar α_i .
<code>a_array</code>	Array, size <code>total_batch_count</code> , of pointers to arrays used to store A matrices.

lda_array

Array of size *group_count*. For the group *i*, *lda_i* = *lda_array[i]* specifies the leading dimension of the array storing matrix *A* as declared in the calling (sub)program.

	<i>transa_i</i> =CblasNoTrans	<i>transa_i</i> =CblasTrans or <i>transa_i</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	<i>lda_i</i> must be at least $\max(1, m_i)$.	<i>lda_i</i> must be at least $\max(1, k_i)$
<i>Layout</i> = CblasRowMajor	<i>lda_i</i> must be at least $\max(1, k_i)$	<i>lda_i</i> must be at least $\max(1, m_i)$.

b_array

Array, size *total_batch_count*, of pointers to arrays used to store *B* matrices.

ldb_array

Array of size *group_count*. For the group *i*, *ldb_i* = *ldb_array[i]* specifies the leading dimension of the array storing matrix *B* as declared in the calling (sub)program.

	<i>transb_i</i> =CblasNoTrans	<i>transb_i</i> =CblasTrans or <i>transb_i</i> =CblasConjTrans
<i>Layout</i> = CblasColMajor	<i>ldb_i</i> must be at least $\max(1, k_i)$.	<i>ldb_i</i> must be at least $\max(1, n_i)$.
<i>Layout</i> = CblasRowMajor	<i>ldb_i</i> must be at least $\max(1, n_i)$.	<i>ldb_i</i> must be at least $\max(1, k_i)$.

beta_array

For the group *i*, *beta_array[i]* specifies the scalar *beta_i*.

When *beta_i* is equal to zero, then *C* matrices in group *i* need not be set on input.

c_array

Array, size *total_batch_count*, of pointers to arrays used to store *C* matrices.

ldc_array

Array of size *group_count*. For the group *i*, *ldc_i* = *ldc_array[i]* specifies the leading dimension of all arrays storing matrix *C* in group *i* as declared in the calling (sub)program.

When *Layout* = CblasColMajor/*ldc_i* must be at least $\max(1, m_i)$.

When *Layout* = CblasRowMajor/*ldc_i* must be at least $\max(1, n_i)$.

group_count

Specifies the number of groups. Must be at least 0.

group_size

Array of size *group_count*. The element *group_size[i]* specifies the number of matrices in group *i*. Each element in *group_size* must be at least 0.

Output Parameters

c_array

Overwritten by the *m_i*-by-*n_i* matrix (*alpha_i**op(*A*)*op(*B*) + *beta_i***C*) for group *i*.

Application Notes

These routines perform a complex matrix multiplication by forming the real and imaginary parts of the input matrices. This uses three real matrix multiplications and five real matrix additions instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications reduces the time spent in matrix operations by 25%, resulting in significant savings in compute time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

$$fl(x \text{ op } y) = (x \text{ op } y) (1 + \delta), |\delta| \leq u, \text{ op} = \times, /, fl(x \pm y) = (x(1 + \alpha) \pm y(1 + \beta)), |\alpha|, |\beta| \leq u$$

then for an n -by- n matrix $\hat{C} = fl(C_1 + iC_2) = fl((A_1 + iA_2)(B_1 + iB_2)) = \hat{C}_1 + i\hat{C}_2$, the following bounds are satisfied:

$$\|\hat{C}_1 - C_1\| \leq 2(n+1)u\|A\|_{\infty}\|B\|_{\infty} + O(u^2),$$

$$\|\hat{C}_2 - C_2\| \leq 4(n+4)u\|A\|_{\infty}\|B\|_{\infty} + O(u^2),$$

where $\|A\|_{\infty} = \max(\|A_1\|_{\infty}, \|A_2\|_{\infty})$, and $\|B\|_{\infty} = \max(\|B_1\|_{\infty}, \|B_2\|_{\infty})$.

Thus the corresponding matrix multiplications are stable.

cblas_?trsm_batch

Solves a triangular matrix equation for a group of matrices.

Syntax

```
void cblas_strsm_batch (const CBLAS_LAYOUT Layout, const CBLAS_SIDE *Side_Array, const
CBLAS_UPLO *Uplo_Array, const CBLAS_TRANSPOSE *Transa_Array, const CBLAS_DIAG
*Diag_Array, const MKL_INT *M_Array, const MKL_INT *N_Array, const float *alpha_Array,
const float * *A_Array, const MKL_INT *lda_Array, float * *B_Array, const MKL_INT
*ldb_Array, const MKL_INT group_count, const MKL_INT *group_size );
```

```
void cblas_dtrsm_batch (const CBLAS_LAYOUT Layout, const CBLAS_SIDE *Side_Array, const
CBLAS_UPLO *Uplo_Array, const CBLAS_TRANSPOSE *Transa_Array, const CBLAS_DIAG
*Diag_Array, const MKL_INT *M_Array, const MKL_INT *N_Array, const double *alpha_Array,
const double * *A_Array, const MKL_INT *lda_Array, double * *B_Array, const MKL_INT
*ldb_Array, const MKL_INT group_count, const MKL_INT *group_size );
```

```
void cblas_ctrsm_batch (const CBLAS_LAYOUT Layout, const CBLAS_SIDE *Side_Array, const
CBLAS_UPLO *Uplo_Array, const CBLAS_TRANSPOSE *Transa_Array, const CBLAS_DIAG
*Diag_Array, const MKL_INT *M_Array, const MKL_INT *N_Array, const void *alpha_Array,
const void * *A_Array, const MKL_INT *lda_Array, void * *B_Array, const MKL_INT
*ldb_Array, const MKL_INT group_count, const MKL_INT *group_size );
```

```
void cblas_ztrsm_batch (const CBLAS_LAYOUT Layout, const CBLAS_SIDE *Side_Array, const
CBLAS_UPLO *Uplo_Array, const CBLAS_TRANSPOSE *Transa_Array, const CBLAS_DIAG
*Diag_Array, const MKL_INT *M_Array, const MKL_INT *N_Array, const void *alpha_Array,
const void * *A_Array, const MKL_INT *lda_Array, void * *B_Array, const MKL_INT
*ldb_Array, const MKL_INT group_count, const MKL_INT *group_size );
```

Include Files

- mkl.h

Description

The `?trsm_batch` routines solve a series of matrix equations. They are similar to the `?trsm` routines except that they operate on groups of matrices which have the same parameters. The `?trsm_batch` routines process a number of groups at once.

```

idx = 0
for i = 0..group_count - 1
  alpha in alpha_array[i]
  for j = 0..group_size[i] - 1
    A and B matrix in a_array[idx] and b_array[idx]
    Solve op(A)*X = alpha*B
    or
    Solve X*op(A) = alpha*B
    idx = idx + 1
  end for
end for

```

where:

alpha is a scalar element of *alpha_array*,

X and *B* are *m*-by-*n* matrices for *m* and *n* which are elements of *m_array* and *n_array*, respectively,

A is a unit, or non-unit, upper or lower triangular matrix,

and *op(A)* is one of *op(A) = A*, or *op(A) = A^T*, or *op(A) = conjg(A^T)*.

A and *B* represent matrices stored at addresses pointed to by *a_array* and *b_array*, respectively. There are *total_batch_count* entries in each of *a_array* and *b_array*, where *total_batch_count* is the sum of all the *group_size* entries.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>side_array</i>	<p>Array of size <i>group_count</i>. For group <i>i</i>, $0 \leq i < \text{group_count} - 1$, <i>side_i</i> = <i>side_array[i]</i> specifies whether <i>op(A)</i> appears on the left or right of <i>X</i> in the equation:</p> <p>if <i>side_i</i> = CblasLeft, then <i>op(A)*X = alpha*B</i>;</p> <p>if <i>side_i</i> = CblasRight, then <i>X*op(A) = alpha*B</i>.</p>
<i>uplo_array</i>	<p>Array of size <i>group_count</i>. For group <i>i</i>, $0 \leq i < \text{group_count} - 1$, <i>uplo_i</i> = <i>uplo_array[i]</i> specifies whether the matrix <i>A</i> is upper or lower triangular:</p> <p><i>uplo_i</i> = CblasUpper</p> <p>if <i>uplo_i</i> = CblasLower, then the matrix is low triangular.</p>
<i>transa_array</i>	<p>Array of size <i>group_count</i>. For group <i>i</i>, $0 \leq i < \text{group_count} - 1$, <i>transa_i</i> = <i>transa_array[i]</i> specifies the form of <i>op(A)</i> used in the matrix multiplication:</p> <p>if <i>transa_i</i>=CblasNoTrans, then <i>op(A) = A</i>;</p> <p>if <i>transa_i</i>=CblasTrans;</p> <p>if <i>transa_i</i>=CblasConjTrans, then <i>op(A) = conjg(A')</i>.</p>
<i>diag_array</i>	<p>Array of size <i>group_count</i>. For group <i>i</i>, $0 \leq i < \text{group_count} - 1$, <i>diag_i</i> = <i>diag_array[i]</i> specifies whether the matrix <i>A</i> is unit triangular:</p> <p>if <i>diag_i</i> = CblasUnit then the matrix is unit triangular;</p>

if $diag_i = \text{CblasNonUnit}$, then the matrix is not unit triangular.

m_array Array of size *group_count*. For group *i*, $0 \leq i < \text{group_count} - 1$, $m_i = m_array[i]$ specifies the number of rows of *B*. The value of m_i must be at least zero.

n_array Array of size *group_count*. For group *i*, $0 \leq i < \text{group_count} - 1$, $n_i = n_array[i]$ specifies the number of columns of *B*. The value of n_i must be at least zero.

alpha_array Array of size *group_count*. For group *i*, $0 \leq i < \text{group_count} - 1$, *alpha_array*[*i*] specifies the scalar α_i .

a_array Array, size *total_batch_count*, of pointers to arrays used to store *A* matrices.

For group *i*, $0 \leq i < \text{group_count} - 1$, *k* is m_i when *side_i* = CblasLeft and is n_i when *side_i* = CblasRight and *a* is any of the *group_size*[*i*] arrays starting with *a_array*[*group_size*[0] + *group_size*[1] + ... + *group_size*(*i* - 1)]:

Before entry with *uplo_i* = CblasUpper, the leading *k* by *k* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo_i* = CblasLower lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced.

When *diag_i* = CblasUnit, the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda_array Array of size *group_count*. For group *i*, $0 \leq i < \text{group_count} - 1$, *lda_i* = *lda_array*[*i*] specifies the leading dimension of *a* as declared in the calling (sub)program. When *side_i* = CblasLeft, then *lda_i* must be at least $\max(1, m_i)$, when *side_i* = CblasRight, then *lda_i* must be at least $\max(1, n_i)$.

b_array Array, size *total_batch_count*, of pointers to arrays used to store *B* matrices.

For group *i*, $0 \leq i < \text{group_count} - 1$, *b* is any of the *group_size*[*i*] arrays starting with *b_array*[*group_size*[0] + *group_size*[1] + ... + *group_size*(*i* - 1)]:

For *Layout* = CblasColMajor: before entry, the leading m_i -by- n_i part of the array *b* must contain the matrix *B*.

For *Layout* = CblasRowMajor: before entry, the leading n_i -by- m_i part of the array *b* must contain the matrix *B*.

ldb_array Array of size *group_count*. Specifies the leading dimension of *b* as declared in the calling (sub)program. When *Layout* = CblasColMajor, *ldb* must be at least $\max(1, m)$; otherwise, *ldb* must be at least $\max(1, n)$.

Array of size *group_count*. For group *i*, $0 \leq i < \text{group_count} - 1$, *ldb_i* = *ldb_array*[*i*] specifies the leading dimension of *b* as declared in the calling (sub)program. When *Layout* = CblasColMajor, *ldb_i* must be at least $\max(1, m_i)$; otherwise, *ldb_i* must be at least $\max(1, n_i)$.

<code>group_count</code>	Specifies the number of groups. Must be at least 0.
<code>group_size</code>	Array of size <code>group_count</code> . The element <code>group_size[i]</code> specifies the number of matrices in group <i>i</i> . Each element in <code>group_size</code> must be at least 0.

Output Parameters

<code>b_array</code>	Overwritten by the solution matrix <i>X</i> .
----------------------	---

cblas_?trsm_batch_strided

Solves groups of triangular matrix equations.

Syntax

```
void cblas_strsm_batch_strided(const CBLAS_LAYOUT layout, const CBLAS_SIDE side, const
CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m,
const MKL_INT n, const float alpha, const float *a, const MKL_INT lda, const MKL_INT
stridea, float *b, const MKL_INT ldb, const MKL_INT strideb, MKL_INT batch_size);
```

```
void cblas_dtrsm_batch_strided(const CBLAS_LAYOUT layout, const CBLAS_SIDE side, const
CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m,
const MKL_INT n, const double alpha, const double *a, const MKL_INT lda, const MKL_INT
stridea, double *b, const MKL_INT ldb, const MKL_INT strideb, const MKL_INT
batch_size);
```

```
void cblas_ctrsm_batch_strided(const CBLAS_LAYOUT layout, const CBLAS_SIDE side, const
CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m,
const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const MKL_INT
stridea, void *b, const MKL_INT ldb, const MKL_INT strideb, const MKL_INT batch_size);
```

```
void zblas_ctrsm_batch_strided(const CBLAS_LAYOUT layout, const CBLAS_SIDE side, const
CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m,
const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const MKL_INT
stridea, void *b, const MKL_INT ldb, const MKL_INT strideb, const MKL_INT batch_size);
```

Include Files

- `mkl.h`

Description

The `cblas_?trsm_batch_strided` routines solve a series of triangular matrix equations. They are similar to the `cblas_?trsm` routine counterparts, but the `cblas_?trsm_batch_strided` routines solve triangular matrix equations with groups of matrices. All matrix *a* have the same parameters (*size*, *leading dimension*, *side*, *uplo*, *diag*, *transpose operation*) and are stored at constant *stridea* from each other. Similarly, all matrix *b* have the same parameters (*size*, *leading dimension*, *alpha scaling*) and are stored at constant *strideb* from each other.

The operation is defined as

Input Parameters

<code>Layout</code>	Specifies whether two-dimensional array storage is row-major (<code>CblasRowMajor</code>) or column-major (<code>CblasColMajor</code>).
<code>side</code>	Specifies whether $op(A)$ appears on the left or right of <i>X</i> in the equation.

	<p>if $side = \text{CblasLeft}$, then $\text{op}(A) * X = \alpha * B$;</p> <p>if $side = \text{CblasRight}$, then $X * \text{op}(A) = \alpha * B$.</p>
<i>uplo</i>	<p>Specifies whether the matrices A are upper or lower triangular.</p> <p>if $uplo = \text{CblasUpper}$, then A are upper triangular;</p> <p>if $uplo = \text{CblasLower}$, then A are lower triangular.</p>
<i>transa</i>	<p>Specifies $\text{op}(A)$ the transposition operation applied to the matrices A.</p> <p>if $transa = \text{CblasNoTrans}$, then $\text{op}(A) = A$;</p> <p>if $transa = \text{CblasTrans}$, then $\text{op}(A) = A^T$;</p> <p>if $transa = \text{CblasConjTrans}$, then $\text{op}(A) = A^H$;</p>
<i>diag</i>	<p>Specifies whether the matrices A are unit triangular.</p> <p>if $diag = \text{CblasUnit}$, then A are unit triangular;</p> <p>if $diag = \text{CblasLower}$, then A are non-unit triangular.</p>
<i>m</i>	Number of rows of B matrices. Must be at least 0
<i>n</i>	Number of columns of B matrices. Must be at least 0
<i>alpha</i>	Specifies the scalar α .
<i>a</i>	<p>Array of size at least $stridea * batch_size$ holding the A matrices. Each A matrix is stored at constant $stridea$ from each other.</p> <p>Each A matrix has size $lda * k$, where k is m when $side = \text{CblasLeft}$ and is n when $side = \text{CblasRight}$.</p> <p>Before entry with $uplo = \text{CblasUpper}$, the leading k-by-k upper triangular part of the array A must contain the upper triangular matrix and the strictly lower triangular part of A is not referenced.</p> <p>Before entry with $uplo = \text{CblasLower}$ lower triangular part of the array A must contain the lower triangular matrix and the strictly upper triangular part of A is not referenced.</p> <p>When $diag = \text{CblasUnit}$, the diagonal elements of A are not referenced either, but are assumed to be unity.</p>
<i>lda</i>	<p>Specifies the leading dimension of the A matrices. When $side = \text{CblasLeft}$, then lda must be at least $\max(1, m)$, when $side = \text{CblasRight}$, then lda must be at least $\max(1, n)$.</p>
<i>stridea</i>	<p>Stride between two consecutive A matrices.</p> <p>When $side = \text{CblasLeft}$, then $stridea$ must be at least $lda * m$.</p> <p>When $side = \text{CblasRight}$, then $stridea$ must be at least $lda * n$.</p>
<i>b</i>	<p>Array of size at least $strideb * batch_size$ holding the B matrices. Each B matrix is stored at constant $strideb$ from each other.</p> <p>When $layout = \text{CblasColMajor}$, each B matrix has size $ldb * n$. Before entry, the leading m-by-n part of the array B must contain the matrix B.</p>

When *layout*= CblasRowMajor, each *B* matrix has size *ldb** *m*. Before entry, the leading *n*-by-*m* part of the array *B* must contain the matrix *B*.

ldb

Specifies the leading dimension of the *B* matrices.

When *layout*= CblasColMajor, *strideb* must be at least $\max(1, m)$.

Otherwise, *strideb* must be at least $\max(1, n)$.

strideb

Stride between two consecutive *B* matrices.

When *layout*= CblasColMajor, *strideb* must be at least *ldb***n*. Otherwise, *strideb* must be at least *ldb***m*.

batch_size

Number of `trsm` computations to perform. Must be at least 0.

Output Parameters

b

Overwritten by the solution *batch_size* *X* matrices.

mkl_?imatcopy

Performs scaling and in-place transposition/copying of matrices.

Syntax

```
void mkl_simatcopy (const char ordering, const char trans, size_t rows, size_t cols,
const float alpha, float * AB, size_t lda, size_t ldb);
```

```
void mkl_dimatcopy (const char ordering, const char trans, size_t rows, size_t cols,
const double alpha, double * AB, size_t lda, size_t ldb);
```

```
void mkl_cimatcopy (const char ordering, const char trans, size_t rows, size_t cols,
const MKL_Complex8 alpha, MKL_Complex8 * AB, size_t lda, size_t ldb);
```

```
void mkl_zimatcopy (const char ordering, const char trans, size_t rows, size_t cols,
const MKL_Complex16 alpha, MKL_Complex16 * AB, size_t lda, size_t ldb);
```

Include Files

- `mkl.h`

Description

The `mkl_?imatcopy` routine performs scaling and in-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

$AB := \alpha * \text{op}(AB)$.

NOTE

Different arrays must not overlap.

Input Parameters

ordering

Ordering of the matrix storage.

	<p>If <code>ordering = 'R'</code> or <code>'r'</code>, the ordering is row-major.</p> <p>If <code>ordering = 'C'</code> or <code>'c'</code>, the ordering is column-major.</p>
<code>trans</code>	<p>Parameter that specifies the operation type.</p> <p>If <code>trans = 'N'</code> or <code>'n'</code>, $op(AB) = AB$ and the matrix <i>AB</i> is assumed unchanged on input.</p> <p>If <code>trans = 'T'</code> or <code>'t'</code>, it is assumed that <i>AB</i> should be transposed.</p> <p>If <code>trans = 'C'</code> or <code>'c'</code>, it is assumed that <i>AB</i> should be conjugate transposed.</p> <p>If <code>trans = 'R'</code> or <code>'r'</code>, it is assumed that <i>AB</i> should be only conjugated.</p> <p>If the data is real, then <code>trans = 'R'</code> is the same as <code>trans = 'N'</code>, and <code>trans = 'C'</code> is the same as <code>trans = 'T'</code>.</p>
<code>rows</code>	The number of rows in matrix <i>AB</i> before the transpose operation.
<code>cols</code>	The number of columns in matrix <i>AB</i> before the transpose operation.
<code>ab</code>	Array.
<code>alpha</code>	This parameter scales the input matrix by <i>alpha</i> .
<code>lda</code>	<p>Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix; measured in the number of elements.</p> <p>This parameter must be at least <i>rows</i> if <code>ordering = 'C'</code> or <code>'c'</code>, and $\max(1, cols)$ otherwise.</p>
<code>ldb</code>	<p>Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix; measured in the number of elements.</p> <p>To determine the minimum value of <i>ldb</i> on output, consider the following guideline:</p> <p>If <code>ordering = 'C'</code> or <code>'c'</code>, then</p> <ul style="list-style-type: none"> • If <code>trans = 'T'</code> or <code>'t'</code> or <code>'C'</code> or <code>'c'</code>, this parameter must be at least $\max(1, cols)$ • If <code>trans = 'N'</code> or <code>'n'</code> or <code>'R'</code> or <code>'r'</code>, this parameter must be at least $\max(1, rows)$ <p>If <code>ordering = 'R'</code> or <code>'r'</code>, then</p> <ul style="list-style-type: none"> • If <code>trans = 'T'</code> or <code>'t'</code> or <code>'C'</code> or <code>'c'</code>, this parameter must be at least $\max(1, rows)$ • If <code>trans = 'N'</code> or <code>'n'</code> or <code>'R'</code> or <code>'r'</code>, this parameter must be at least $\max(1, cols)$

Output Parameters

<code>ab</code>	<p>Array.</p> <p>Contains the matrix <i>AB</i>.</p>
-----------------	---

Application Notes

For threading to be active in `mkl_?imatcopy`, the pointer *AB* must be aligned on the 64-byte boundary. This requirement can be met by allocating *AB* with `mkl_malloc`.

Interfaces

`mkl_?imatcopy_batch`

Computes a group of in-place scaled matrix copy or transposition operations on general matrices.

Syntax

```
void mkl_simatcopy_batch (char layout, const char * trans_array, const size_t *
rows_array, const size_t * cols_array, const float * alpha_array, float ** ab_array,
const size_t * lda_array, const size_t * ldb_array, size_t group_count, const size_t *
group_size);

void mkl_dimatcopy_batch (char layout, const char * trans_array, const size_t *
rows_array, const size_t * cols_array, const double * alpha_array, double ** ab_array,
const size_t * lda_array, const size_t * ldb_array, size_t group_count, const size_t *
group_size);

void mkl_cimatcopy_batch (char layout, const char * trans_array, const size_t *
rows_array, const size_t * cols_array, const MKL_Complex8 * alpha_array, MKL_Complex8 **
ab_array, const size_t * lda_array, const size_t * ldb_array, size_t group_count, const
size_t * group_size);

void mkl_zimatcopy_batch (char layout, const char * trans_array, const size_t *
rows_array, const size_t * cols_array, const MKL_Complex16 * alpha_array, MKL_Complex16
** ab_array, const size_t * lda_array, const size_t * ldb_array, size_t group_count,
const size_t * group_size);
```

Description

The `mkl_?imatcopy_batch` routine performs a series of in-place scaled matrix copies or transpositions. They are similar to the `mkl_?imatcopy` routine counterparts, but the `mkl_?imatcopy_batch` routine performs matrix operations with groups of matrices. Each group has the same parameters (matrix size, leading dimension, and scaling parameter), but a single call to `mkl_?imatcopy_batch` operates on multiple groups, and each group can have different parameters, unlike the related `mkl_?imatcopy_batch_strided` routines.

The operation is defined as

```
idx = 0
for i = 0..group_count - 1
  m in rows_array[i], n in cols_array[i], and alpha in alpha_array[i]
  for j = 0..group_size[i] - 1
    AB matrices in AB_array[idx]
    AB := alpha*op(AB)
    idx = idx + 1
  end for
end for
```

Where $\text{op}(X)$ is one of $\text{op}(X)=X$, $\text{op}(X)=X'$, $\text{op}(X)=\text{conjg}(X')$, or $\text{op}(X)=\text{conjg}(X)$. On entry, *AB* is a *m*-by-*n* matrix such that *m* and *n* are elements of *rows_array* and *cols_array*.

AB represents a matrix stored at addresses pointed to by *AB_array*. The number of entries in *AB_array* is *total_batch_count* = the sum of all of the *group_size* entries.

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (R) or column-major (C).
<i>trans_array</i>	<p>Array of size <i>group_count</i>. For the group <i>i</i>, <i>trans</i> = <i>trans_array</i>[<i>i</i>] specifies the form of $\text{op}(AB)$, the transposition operation applied to the <i>AB</i> matrix:</p> <p>If <i>trans</i> = 'N' or 'n', $\text{op}(AB) = AB$.</p> <p>If <i>trans</i> = 'T' or 't', $\text{op}(AB) = AB'$</p> <p>If <i>trans</i> = 'C' or 'c', $\text{op}(AB) = \text{conjg}(AB')$</p> <p>If <i>trans</i> = 'R' or 'r', $\text{op}(AB) = \text{conjg}(AB)$</p>
<i>rows_array</i>	Array of size <i>group_count</i> . Specifies the number of rows of the input matrix <i>AB</i> . The value of each element must be at least zero.
<i>cols_array</i>	Array of size <i>group_count</i> . Specifies the number of columns of the input matrix <i>AB</i> . The value of each element must be at least zero.
<i>alpha_array</i>	Array of size <i>group_count</i> . Specifies the scalar <i>alpha</i> .
<i>AB_array</i>	Array of size <i>total_batch_count</i> , holding pointers to arrays used to store <i>AB</i> matrices.
<i>lda_array</i>	Array of size <i>group_count</i> . The leading dimension of the matrix input <i>AB</i> . It must be positive and at least <i>m</i> if column major layout is used or at least <i>n</i> if row major layout is used.
<i>ldb_array</i>	<p>Array of size <i>group_count</i>. The leading dimension of the matrix input <i>AB</i>. It must be positive and at least</p> <p><i>m</i> if column major layout is used and $\text{op}(AB) = AB$ or $\text{conjg}(AB)$</p> <p><i>n</i> if row major layout is used and $\text{op}(AB) = AB'$ or $\text{conjg}(AB')$</p> <p><i>n</i> otherwise</p>
<i>group_count</i>	Specifies the number of groups. Must be at least 0
<i>group_size</i>	Array of size <i>group_count</i> . The element <i>group_size</i> [<i>i</i>] specifies the number of matrices in group <i>i</i> . Each element in <i>group_size</i> must be at least 0.

Output Parameters

<i>AB_array</i>	Output array of size <i>total_batch_count</i> , holding pointers to arrays used to store the updated <i>AB</i> matrices.
-----------------	--

mkl_?imatcopy_batch_strided

Computes a group of in-place scaled matrix copy or transposition using general matrices.

Syntax

```
void mkl_simatcopy_batch_strided (const char layout, const char trans, size_t row,
size_t col, const float alpha, float * ab, size_t lda, size_t ldb, size_t stride, size_t
batch_size);
```

```
void mkl_dimatcopy_batch_strided (const char layout, const char trans, size_t row,
size_t col, const double alpha, double * ab, size_t lda, size_t ldb, size_t stride,
size_t batch_size);
```

```
void mkl_cimatcopy_batch_strided (const char layout, const char trans, size_t row,
size_t col, MKL_complex8 alpha, MKL_complex8 * ab, size_t lda, size_t ldb, size_t
stride, size_t batch_size);
```

```
void mkl_zimatcopy_batch_strided (const char layout, const char trans, size_t row,
size_t col, MKL_complex16 alpha, MKL_complex16 * ab, size_t lda, size_t ldb, size_t
stride, size_t batch_size);
```

Description

The `mkl_?imatcopy_batch_strided` routine performs a series of scaled matrix copy or transposition. They are similar to the `mkl_?imatcopy` routine counterparts, but the `mkl_?imatcopy_batch_strided` routine performs matrix operations with a group of matrices.

All matrices `ab` have the same parameters (size, transposition operation...) and are stored at constant stride from each other. The operation is defined as

```
for i = 0 ... batch_size - 1
    AB is a matrix at offset i * stride in ab
    AB = alpha * op(AB)
end for
```

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor)
<i>trans</i>	Specifies <code>op(AB)</code> , the transposition operation applied to the <code>AB</code> matrices. If <code>trans = 'N' or 'n'</code> , <code>op(AB) = AB</code> . If <code>trans = 'T' or 't'</code> , <code>op(AB) = AB'</code> If <code>trans = 'C' or 'c'</code> , <code>op(AB) = conjg(AB')</code> If <code>trans = 'R' or 'r'</code> , <code>op(AB) = conjg(AB)</code>
<i>row</i>	Specifies the number of rows of the matrices <code>AB</code> . The value of <i>row</i> must be at least zero.
<i>col</i>	Specifies the number of columns of the matrices <code>AB</code> . The value of <i>col</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>ab</i>	Array holding all the input matrix <code>AB</code> . Must be of size at least <code>batch_size * stride</code> .
<i>lda</i>	The leading dimension of the matrix input <code>AB</code> . It must be positive and at least <i>row</i> if column major layout is used or at least <i>col</i> if row major layout is used.

<i>ldb</i>	<p>The leading dimension of the matrix input AB. It must be positive and at least</p> <p><i>row</i> if column major layout is used and $\text{op}(AB) = AB$ or $\text{conjg}(AB)$</p> <p><i>row</i> if row major layout is used and $\text{op}(AB) = AB'$ or $\text{conjg}(AB')$</p> <p><i>col</i> otherwise</p>
<i>stride</i>	<p>Stride between two consecutive AB matrices, must be at least $\max(ldb, lda) * \max(ka, kb)$ where</p> <ul style="list-style-type: none"> • <i>ka</i> is <i>row</i> if column major layout is used or <i>col</i> if row major layout is used • <i>kb</i> is <i>col</i> if column major layout is used and $\text{op}(AB) = AB$ or $\text{conjg}(AB)$ or row major layout is used and $\text{op}(AB) = AB'$ or $\text{conjg}(AB')$; <i>kb</i> is <i>row</i> otherwise.
<i>batch_size</i>	<p>Number of <i>imatcopy</i> computations to perform and AB matrices. Must be at least 0.</p>

Output Parameters

<i>ab</i>	Array holding the <i>batch_size</i> updated matrices AB.
-----------	--

mkl_omatadd_batch_strided

Computes a group of out-of-place scaled matrix additions using general matrices.

Syntax

```
void mkl_somatadd_batch_strided(char ordering, char transa, char transb, size_t rows,
size_t cols, float alpha, const float * A, size_t lda, size_t stridea, float beta,
const float * B, size_t ldb, size_t strideb, float * C, size_t ldc, size_t stridec,
size_t batch_size);
```

```
void mkl_domatadd_batch_strided(char ordering, char transa, char transb, size_t rows,
size_t cols, double alpha, const double * A, size_t lda, size_t stridea, double beta,
const double * B, size_t ldb, size_t strideb, double * C, size_t ldc, size_t stridec,
size_t batch_size);
```

```
void mkl_comatadd_batch_strided(char ordering, char transa, char transb, size_t rows,
size_t cols, MKL_Complex8 alpha, const MKL_Complex8 * A, size_t lda, size_t stridea,
MKL_Complex8 beta, const MKL_Complex8 * B, size_t ldb, size_t strideb, MKL_Complex8 *
C, size_t ldc, size_t stridec, size_t batch_size);
```

```
void mkl_zomatadd_batch_strided(char ordering, char transa, char transb, size_t rows,
size_t cols, MKL_Complex16 alpha, const MKL_Complex16 * A, size_t lda, size_t stridea,
MKL_Complex16 beta, const MKL_Complex16 * B, size_t ldb, size_t strideb, MKL_Complex16
* C, size_t ldc, size_t stridec, size_t batch_size);
```

Description

The `mkl_omatadd_batch_strided` routines perform a series of scaled matrix additions. They are similar to the `mkl_omatadd` routines, but the `mkl_omatadd_batch_strided` routines perform matrix operations with a group of matrices.

The matrices A, B, and C are stored at a constant stride from each other in memory, given by the parameters `stridea`, `strideb`, and `stridec`. The operation is defined as:

```
for i = 0 ... batch_size - 1
  A is a matrix at offset i * stridea in the array a
  B is a matrix at offset i * strideb in the array b
  C is a matrix at offset i * stridec in the array c
  C = alpha * op(A) + beta * op(B)
end for
```

where:

- `op(X)` is one of `op(X) = X`, `op(X) = X'`, `op(X) = conjg(X)` or `op(X) = conjg(X')`.
- `alpha` and `beta` are scalars.
- A, B, and C are matrices.

The input arrays `a` and `b` contain all the input matrices, and the single output array `c` contains all the output matrices. The locations of the individual matrices within the array are given by stride lengths, while the number of matrices is given by the `batch_size` parameter.

Input Parameters

layout	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
transa	Specifies <code>op(A)</code> , the transposition operation applied to the matrices A. 'N' or 'n' indicates no operation, 'T' or 't' is transposition, 'R' or 'r' is complex conjugation without transposition, and 'C' or 'c' is conjugate transposition.
transb	Specifies <code>op(B)</code> , the transposition operation applied to the matrices B.
rows	Number of rows for the result matrix C. Must be at least zero.
cols	Number of columns for the result matrix C. Must be at least zero.
alpha	Scaling factor for the matrices A.
a	Array holding the input matrices A. Must have size at least <code>stride_a*batch_size</code> .
lda	Leading dimension of the A matrices. If matrices are stored using column major layout, <code>lda</code> must be at least <code>rows</code> if A is not transposed or <code>cols</code> if A is transposed. If matrices are stored using row major layout, <code>lda</code> must be at least <code>cols</code> if A is not transposed or at least <code>rows</code> if A is transposed. Must be positive.
stride_a	Stride between the different A matrices. If matrices are stored using column major layout, <code>stride_a</code> must be at least <code>lda*rows</code> if A is not transposed or at least <code>lda*cols</code> if A is transposed. If matrices are stored using row major layout, <code>stride_a</code> must be at least <code>lda*rows</code> if B is not transposed or at least <code>lda*cols</code> if A is transposed.
beta	Scaling factor for the matrices B.
b	Array holding the input matrices B. Must have size at least <code>stride_b*batch_size</code> .

<code>ldb</code>	Leading dimension of the B matrices. If matrices are stored using column major layout, <code>ldb</code> must be at least <code>rows</code> if B is not transposed or <code>cols</code> if B is transposed. If matrices are stored using row major layout, <code>ldb</code> must be at least <code>cols</code> if B is not transposed or at least <code>rows</code> if B is transposed. Must be positive.
<code>stride_b</code>	Stride between the different B matrices. If matrices are stored using column major layout, <code>stride_b</code> must be at least <code>ldb*cols</code> if B is not transposed or at least <code>ldb*rows</code> if B is transposed. If matrices are stored using row major layout, <code>stride_b</code> must be at least <code>ldb*rows</code> if B is not transposed or at least <code>ldb*cols</code> if B is transposed.
<code>c</code>	Output array, overwritten by <code>batch_size</code> matrix addition operations of the form $\alpha * op(A) + \beta * op(B)$. Must have size at least <code>stride_c*batch_size</code> .
<code>lda</code>	Leading dimension of the A matrices. If matrices are stored using column major layout, <code>lda</code> must be at least <code>rows</code> . If matrices are stored using row major layout, <code>lda</code> must be at least <code>cols</code> . Must be positive.
<code>stride_c</code>	Stride between the different C matrices. If matrices are stored using column major layout, <code>stride_c</code> must be at least <code>ldc*cols</code> . If matrices are stored using row major layout, <code>stride_c</code> must be at least <code>ldc*rows</code> .
<code>batch_size</code>	Specifies the number of input and output matrices to add.

Output Parameters

<code>c</code>	Array holding the updated matrices <code>c</code> .
----------------	---

`mkl_?omatcopy`

Performs scaling and out-place transposition/copying of matrices.

Syntax

```
void mkl_somatcopy (char ordering, char trans, size_t rows, size_t cols, const float
alpha, const float * A, size_t lda, float * B, size_t ldb);

void mkl_domatcopy (char ordering, char trans, size_t rows, size_t cols, const double
alpha, const double * A, size_t lda, double * B, size_t ldb);

void mkl_comatcopy (char ordering, char trans, size_t rows, size_t cols, const
MKL_Complex8 alpha, const MKL_Complex8 * A, size_t lda, MKL_Complex8 * B, size_t ldb);

void mkl_zomatcopy (char ordering, char trans, size_t rows, size_t cols, const
MKL_Complex16 alpha, const MKL_Complex16 * A, size_t lda, MKL_Complex16 * B, size_t
ldb);
```

Include Files

- `mkl.h`

Description

The `mkl_omatcopy` routine performs scaling and out-of-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

$$B := \alpha * \text{op}(A)$$

NOTE

Different arrays must not overlap.

Input Parameters

<i>ordering</i>	Ordering of the matrix storage. If <i>ordering</i> = 'R' or 'r', the ordering is row-major. If <i>ordering</i> = 'C' or 'c', the ordering is column-major.
<i>trans</i>	Parameter that specifies the operation type. If <i>trans</i> = 'N' or 'n', $\text{op}(A) = A$ and the matrix <i>A</i> is assumed unchanged on input. If <i>trans</i> = 'T' or 't', it is assumed that <i>A</i> should be transposed. If <i>trans</i> = 'C' or 'c', it is assumed that <i>A</i> should be conjugate transposed. If <i>trans</i> = 'R' or 'r', it is assumed that <i>A</i> should be only conjugated. If the data is real, then <i>trans</i> = 'R' is the same as <i>trans</i> = 'N', and <i>trans</i> = 'C' is the same as <i>trans</i> = 'T'.
<i>rows</i>	The number of rows in matrix <i>A</i> (the input matrix).
<i>cols</i>	The number of columns in matrix <i>A</i> (the input matrix).
<i>alpha</i>	This parameter scales the input matrix by <i>alpha</i> .
<i>a</i>	Input array. If <i>ordering</i> = 'R' or 'r', the size of <i>a</i> is <i>lda</i> * <i>rows</i> . If <i>ordering</i> = 'C' or 'c', the size of <i>a</i> is <i>lda</i> * <i>cols</i> .
<i>lda</i>	If <i>ordering</i> = 'R' or 'r', <i>lda</i> represents the number of elements in array <i>a</i> between adjacent rows of matrix <i>A</i> ; <i>lda</i> must be at least equal to the number of columns of matrix <i>A</i> . If <i>ordering</i> = 'C' or 'c', <i>lda</i> represents the number of elements in array <i>a</i> between adjacent columns of matrix <i>A</i> ; <i>lda</i> must be at least equal to the number of row in matrix <i>A</i> .
<i>b</i>	Output array. If <i>ordering</i> = 'R' or 'r'; <ul style="list-style-type: none"> If <i>trans</i> = 'T' or 't' or 'C' or 'c', the size of <i>b</i> is <i>ldb</i> * <i>cols</i>. If <i>trans</i> = 'N' or 'n' or 'R' or 'r', the size of <i>b</i> is <i>ldb</i> * <i>rows</i>. If <i>ordering</i> = 'C' or 'c';

- If *trans* = 'T' or 't' or 'C' or 'c', the size of *b* is *ldb* * *rows*.
- If *trans* = 'N' or 'n' or 'R' or 'r', the size of *b* is *ldb* * *cols*.

ldb

If *ordering* = 'R' or 'r', *ldb* represents the number of elements in array *b* between adjacent rows of matrix *B*.

- If *trans* = 'T' or 't' or 'C' or 'c', *ldb* must be at least equal to *rows*.
- If *trans* = 'N' or 'n' or 'R' or 'r', *ldb* must be at least equal to *cols*.

If *ordering* = 'C' or 'c', *ldb* represents the number of elements in array *b* between adjacent columns of matrix *B*.

- If *trans* = 'T' or 't' or 'C' or 'c', *ldb* must be at least equal to *cols*.
- If *trans* = 'N' or 'n' or 'R' or 'r', *ldb* must be at least equal to *rows*.

Output Parameters

b

Output array.

Contains the destination matrix.

Interfaces

mkl_?omatcopy_batch

Computes a group of out of place scaled matrix copy or transposition operations on general matrices.

Syntax

```
void mkl_somatcopy_batch (char layout, const char * trans_array, const size_t *
rows_array, const size_t * cols_array, const float * alpha_array, float ** A_array,
const size_t * lda_array, float ** B_array, const size_t * ldb_array, size_t
group_count, const size_t * group_size);
```

```
void mkl_domatcopy_batch (char layout, const char * trans_array, const size_t *
rows_array, const size_t * cols_array, const double * alpha_array, float ** A_array,
const size_t * lda_array, double ** B_array, const size_t * ldb_array, size_t
group_count, const size_t * group_size);
```

```
void mkl_comatcopy_batch (char layout, const char * trans_array, const size_t *
rows_array, const size_t * cols_array, const MKL_Complex8 * alpha_array, MKL_Complex8 **
A_array, const size_t * lda_array, MKL_Complex8 ** B_array, const size_t * ldb_array,
size_t group_count, const size_t * group_size);
```

```
void mkl_zomatcopy_batch (char layout, const char * trans_array, const size_t *
rows_array, const size_t * cols_array, const MKL_Complex16 * alpha_array, MKL_Complex16
** A_array, const size_t * lda_array, MKL_Complex16 ** B_array, const size_t *
ldb_array, size_t group_count, const size_t * group_size);
```

Description

The `mkl_?omatcopy_batch` routine performs a series of out-of-place scaled matrix copies or transpositions. They are similar to the `mkl_?omatcopy` routine counterparts, but the `mkl_?omatcopy_batch` routine performs matrix operations with groups of matrices. Each group has the same parameters (matrix size, leading dimension, and scaling parameter), but a single call to `mkl_?omatcopy_batch` operates on multiple groups, and each group can have different parameters, unlike the related `mkl_?omatcopy_batch_strided` routines.

The operation is defined as

```
idx = 0
for i = 0..group_count - 1
    m in rows_array[i], n in cols_array[i], and alpha in alpha_array[i]
    for j = 0..group_size[i] - 1
        A and B matrices in a_array[idx] and b_array[idx], respectively
        B := alpha*op(A)
        idx = idx + 1
    end for
end for
```

Where `op(X)` is one of `op(X)=X`, `op(X)=X'`, `op(X)=conjg(X')`, or `op(X)=conjg(X)`. *A* is a *m*-by-*n* matrix such that *m* and *n* are elements of `rows_array` and `cols_array`.

A and *B* represent matrices stored at addresses pointed to by `A_array` and `B_array`. The number of entries in `A_array` and `B_array` is `total_batch_count` = the sum of all of the `group_size` entries.

Input Parameters

<code>layout</code>	Specifies whether two-dimensional array storage is row-major (R) or column-major (C).
<code>trans_array</code>	<p>Array of size <code>group_count</code>. For the group <i>i</i>, <code>trans = trans_array[i]</code> specifies the form of <code>op(A)</code>, the transposition operation applied to the <i>A</i> matrix:</p> <p>If <code>trans = 'N' or 'n'</code>, <code>op(A)=A</code>.</p> <p>If <code>trans = 'T' or 't'</code>, <code>op(A)=A'</code></p> <p>If <code>trans = 'C' or 'c'</code>, <code>op(A)=conjg(A')</code></p> <p>If <code>trans = 'R' or 'r'</code>, <code>op(A)=conjg(A)</code></p>
<code>rows_array</code>	Array of size <code>group_count</code> . Specifies the number of rows of the matrix <i>A</i> . The value of each element must be at least zero.
<code>cols_array</code>	Array of size <code>group_count</code> . Specifies the number of columns of the matrix <i>A</i> . The value of each element must be at least zero.
<code>alpha_array</code>	Array of size <code>group_count</code> . Specifies the scalar <i>alpha</i> .
<code>A_array</code>	Array of size <code>total_batch_count</code> , holding pointers to arrays used to store <i>A</i> input matrices.
<code>lda_array</code>	Array of size <code>group_count</code> . The leading dimension of the input matrix <i>A</i> . It must be positive and at least <i>m</i> if column major layout is used or at least <i>n</i> if row major layout is used.

<code>ldb_array</code>	<p>Array of size <code>group_count</code>. The leading dimension of the output matrix <code>B</code>. It must be positive and at least</p> <p>m if column major layout is used and <code>op(A) = A</code> or <code>conjg(A)</code></p> <p>n if row major layout is used and <code>op(A) = A'</code> or <code>conjg(A')</code></p> <p>n otherwise</p>
<code>group_count</code>	Specifies the number of groups. Must be at least 0
<code>group_size</code>	<p>Array of size <code>group_count</code>. The element <code>group_size[i]</code> specifies the number of matrices in group i. Each element in <code>group_size</code> must be at least 0.</p>

Output Parameters

<code>B_array</code>	Output array of size <code>total_batch_count</code> , holding pointers to arrays used to store the <code>B</code> output matrices, the contents of which are overwritten by the operation of the form <code>alpha*op(A)</code> .
----------------------	--

mkl_?omatcopy_batch_strided

Computes a group of out of place scaled matrix copy or transposition using general matrices.

Syntax

```
void mkl_somatcopy_batch_strided (const char layout, const char trans, size_t row,
size_t col, const float alpha, const float * a, size_t lda, size_t stridea, float * b,
size_t ldb, size_t strideb, size_t batch_size);

void mkl_domatcopy_batch_strided (const char layout, const char trans, size_t row,
size_t col, const double alpha, const double * a, size_t lda, size_t stridea, double *
b, size_t ldb, size_t strideb, size_t batch_size);

void mkl_comatcopy_batch_strided (const char layout, const char trans, size_t row,
size_t col, const MKL_complex8 alpha, const MKL_complex8 * a, size_t lda, size_t
stridea, MKL_complex8 * b, size_t ldb, size_t strideb, size_t batch_size);

void mkl_zomatcopy_batch_strided (const char layout, const char trans, size_t row,
size_t col, const MKL_complex16 alpha, const MKL_complex16 * a, size_t lda, size_t
stridea, MKL_complex16 * b, size_t ldb, size_t strideb, size_t batch_size);
```

Description

The `mkl_?omatcopy_batch_strided` routine performs a series of out-of-place scaled matrix copy or transposition. They are similar to the `mkl_?omatcopy` routine counterparts, but the `mkl_?omatcopy_batch_strided` routine performs matrix operations with group of matrices.

All matrices `a` and `b` have the same parameters (size, transposition operation...) and are stored at constant stride from each other respectively given by `stridea` and `strideb`. The operation is defined as

```
for i = 0 ... batch_size - 1
    A and B are matrices at offset i * stridea in a and I * strideb in b
    B = alpha * op(A)
end for
```

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>trans</i>	Specifies $\text{op}(A)$, the transposition operation applied to the A matrices. If <i>trans</i> = 'N' or 'n', $\text{op}(A)=A$. If <i>trans</i> = 'T' or 't', $\text{op}(A)=A'$ If <i>trans</i> = 'C' or 'c', $\text{op}(A)=\text{conjg}(A')$ If <i>trans</i> = 'R' or 'r', $\text{op}(A)=\text{conjg}(A)$
<i>row</i>	Specifies the number of rows of the matrices A and B . The value of <i>row</i> must be at least zero.
<i>col</i>	Specifies the number of columns of the matrices A and B . The value of <i>col</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>a</i>	Array holding all the input matrices A . Must be of size at least $\text{lda} * k + \text{stridea} * (\text{batch_size} - 1) * \text{stridea}$ where k is <i>col</i> if column major is used and <i>row</i> otherwise.
<i>lda</i>	The leading dimension of the matrix input A . It must be positive and at least <i>row</i> if column major layout is used or at least <i>col</i> if row major layout is used.
<i>stridea</i>	Stride between two consecutive A matrices, must be at least 0.
<i>b</i>	Array holding all the output matrices B . Must be of size at least $\text{batch_size} * \text{strideb}$. The <i>b</i> array must be independent from the <i>a</i> array.
<i>ldb</i>	The leading dimension of the output matrix B . It must be positive and at least: <ul style="list-style-type: none"> • <i>row</i> if column major layout is used and $\text{op}(A) = A$ or $\text{conjg}(A)$ • <i>row</i> if row major layout is used and $\text{op}(A) = A'$ or $\text{conjg}(A')$ • <i>col</i> otherwise
<i>strideb</i>	Stride between two consecutive B matrices. It must be positive and at least: <ul style="list-style-type: none"> • $\text{ldb} * \text{col}$ if column major layout is used and $\text{op}(A) = A$ or $\text{conjg}(A)$ • $\text{ldb} * \text{col}$ if row major layout is used and $\text{op}(A) = A'$ or $\text{conjg}(A')$ • $\text{ldb} * \text{row}$ otherwise
<i>batch_size</i>	

Output Parameters

<i>b</i>	Array holding the <i>batch_size</i> updated matrices B .
----------	--

mkl_?omatcopy2

Performs two-strided scaling and out-of-place transposition/copying of matrices.

Syntax

```
void mkl_somatcopy2 (char ordering, char trans, size_t rows, size_t cols, const float
alpha, const float * A, size_t lda, size_t stridea, float * B, size_t ldb, size_t
strideb);
```

```
void mkl_domatcopy2 (char ordering, char trans, size_t rows, size_t cols, const double
alpha, const double * A, size_t lda, size_t stridea, double * B, size_t ldb, size_t
strideb);
```

```
void mkl_comatcopy2 (char ordering, char trans, size_t rows, size_t cols, const
MKL_Complex8 alpha, const MKL_Complex8 * A, size_t lda, size_t stridea, MKL_Complex8 *
B, size_t ldb, size_t strideb);
```

```
void mkl_zomatcopy2 (char ordering, char trans, size_t rows, size_t cols, const
MKL_Complex16 alpha, const MKL_Complex16 * A, size_t lda, size_t stridea, MKL_Complex16
* B, size_t ldb, size_t strideb);
```

Include Files

- mkl.h

Description

The `mkl_?omatcopy2` routine performs two-strided scaling and out-of-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

$$B := \alpha * \text{op}(A)$$

Normally, matrices in the BLAS or LAPACK are specified by a single stride index. For instance, in the column-major order, $A(2,1)$ is stored in memory one element away from $A(1,1)$, but $A(1,2)$ is a leading dimension away. The leading dimension in this case is at least the number of rows of the source matrix. If a matrix has two strides, then both $A(2,1)$ and $A(1,2)$ may be an arbitrary distance from $A(1,1)$.

NOTE

Different arrays must not overlap.

Input Parameters

ordering

Ordering of the matrix storage.

If *ordering* = 'R' or 'r', the ordering is row-major.

If *ordering* = 'C' or 'c', the ordering is column-major.

trans

Parameter that specifies the operation type.

If *trans* = 'N' or 'n', $\text{op}(A) = A$ and the matrix *A* is assumed unchanged on input.

If *trans* = 'T' or 't', it is assumed that *A* should be transposed.

If `trans = 'C'` or `'c'`, it is assumed that *A* should be conjugate transposed.

If `trans = 'R'` or `'r'`, it is assumed that *A* should be only conjugated.

If the data is real, then `trans = 'R'` is the same as `trans = 'N'`, and `trans = 'C'` is the same as `trans = 'T'`.

`rows`

number of rows for the input matrix *A*. Must be at least zero.

`cols`

Number of columns for the input matrix *A*. Must be at least zero.

`alpha`

Scaling factor for the matrix transposition or copy.

`a`

Array holding the input matrix *A*. Must have size at least *lda* * *n* for column major ordering and at least *lda* * *m* for row major ordering.

`lda`

Leading dimension of the matrix *A*. If matrices are stored using column major layout, *lda* is the number of elements in the array between adjacent columns of the matrix and must be at least `stridea * (m-1) + 1`. If using row major layout, *lda* is the number of elements between adjacent rows of the matrix and must be at least `stridea * (n-1) + 1`.

`stridea`

The second stride of the matrix *A*. For column major layout, *stridea* is the number of elements in the array between adjacent rows of the matrix. For row major layout *stridea* is the number of elements between adjacent columns of the matrix. In both cases *stridea* must be at least 1.

`b`

Array holding the output matrix *B*.

	<code>trans = transpose::nontrans</code>	<code>trans = transpose::trans, or trans = transpose::conjtrans</code>
Column major	<i>B</i> is <i>m</i> x <i>n</i> matrix. Size of array <i>b</i> must be at least <i>ldb</i> * <i>n</i> .	<i>B</i> is <i>n</i> x <i>m</i> matrix. Size of array <i>b</i> must be at least <i>ldb</i> * <i>m</i> .
Row major	<i>B</i> is <i>m</i> x <i>n</i> matrix. Size of array <i>b</i> must be at least <i>ldb</i> * <i>m</i> .	<i>B</i> is <i>n</i> x <i>m</i> matrix. Size of array <i>b</i> must be at least <i>ldb</i> * <i>n</i> .

`ldb`

The leading dimension of the matrix *B*. Must be positive.

	<code>trans = transpose::nontrans</code>	<code>trans = transpose::trans, or trans = transpose::conjtrans</code>
Column major	<i>ldb</i> must be at least <code>strideb * (m-1) + 1</code> .	<i>ldb</i> must be at least <code>strideb * (n-1) + 1</code> .
Row major	<i>ldb</i> must be at least <code>strideb * (n-1) + 1</code> .	<i>ldb</i> must be at least <code>strideb * (m-1) + 1</code> .

`strideb`

The second stride of the matrix *B*. For column major layout, *strideb* is the number of elements in the array between adjacent rows of the matrix. For row major layout, *strideb* is the number of elements between adjacent columns of the matrix. In both cases *strideb* must be at least 1.

Output Parameters

b Array, size at least *m*.
Contains the destination matrix.

Interfaces

mkl_?omatadd

Scales and sums two matrices including in addition to performing out-of-place transposition operations.

Syntax

```
void mkl_somatadd (char ordering, char transa, char transb, size_t m, size_t n, const
float alpha, const float * A, size_t lda, const float beta, const float * B, size_t ldb,
float * C, size_t ldc);
```

```
void mkl_domatadd (char ordering, char transa, char transb, size_t m, size_t n, const
double alpha, const double * A, size_t lda, const double beta, const double * B, size_t
ldb, double * C, size_t ldc);
```

```
void mkl_comatadd (char ordering, char transa, char transb, size_t m, size_t n, const
MKL_Complex8 alpha, const MKL_Complex8 * A, size_t lda, const MKL_Complex8 beta, const
MKL_Complex8 * B, size_t ldb, MKL_Complex8 * C, size_t ldc);
```

```
void mkl_zomatadd (char ordering, char transa, char transb, size_t m, size_t n, const
MKL_Complex16 alpha, const MKL_Complex16 * A, size_t lda, const MKL_Complex16 beta,
const MKL_Complex16 * B, size_t ldb, MKL_Complex16 * C, size_t ldc);
```

Include Files

- mkl.h

Description

The `mkl_?omatadd` routine scales and adds two matrices, as well as performing out-of-place transposition operations. A transposition operation can be no operation, a transposition, a conjugate transposition, or a conjugation (without transposition). The following out-of-place memory movement is done:

$$C := \alpha * \text{op}(A) + \beta * \text{op}(B)$$

where the `op(A)` and `op(B)` operations are transpose, conjugate-transpose, conjugate (no transpose), or no transpose, depending on the values of *transa* and *transb*. If no transposition of the source matrices is required, *m* is the number of rows and *n* is the number of columns in the source matrices *A* and *B*. In this case, the output matrix *C* is *m*-by-*n*.

NOTE

Note that different arrays must not overlap.

Input Parameters

<i>ordering</i>	<p>Ordering of the matrix storage.</p> <p>If <i>ordering</i> = 'R' or 'r', the ordering is row-major.</p> <p>If <i>ordering</i> = 'C' or 'c', the ordering is column-major.</p>
<i>transa</i>	<p>Parameter that specifies the operation type on matrix <i>A</i>.</p> <p>If <i>transa</i> = 'N' or 'n', $\text{op}(A)=A$ and the matrix <i>A</i> is assumed unchanged on input.</p> <p>If <i>transa</i> = 'T' or 't', it is assumed that <i>A</i> should be transposed.</p> <p>If <i>transa</i> = 'C' or 'c', it is assumed that <i>A</i> should be conjugate transposed.</p> <p>If <i>transa</i> = 'R' or 'r', it is assumed that <i>A</i> should be conjugated (and not transposed).</p> <p>If the data is real, then <i>transa</i> = 'R' is the same as <i>transa</i> = 'N', and <i>transa</i> = 'C' is the same as <i>transa</i> = 'T'.</p>
<i>transb</i>	<p>Parameter that specifies the operation type on matrix <i>B</i>.</p> <p>If <i>transb</i> = 'N' or 'n', $\text{op}(B)=B$ and the matrix <i>B</i> is assumed unchanged on input.</p> <p>If <i>transb</i> = 'T' or 't', it is assumed that <i>B</i> should be transposed.</p> <p>If <i>transb</i> = 'C' or 'c', it is assumed that <i>B</i> should be conjugate transposed.</p> <p>If <i>transb</i> = 'R' or 'r', it is assumed that <i>B</i> should be conjugated (and not transposed).</p> <p>If the data is real, then <i>transb</i> = 'R' is the same as <i>transb</i> = 'N', and <i>transb</i> = 'C' is the same as <i>transb</i> = 'T'.</p>
<i>m</i>	The number of matrix rows in $\text{op}(A)$, $\text{op}(B)$, and <i>C</i> .
<i>n</i>	The number of matrix columns in $\text{op}(A)$, $\text{op}(B)$, and <i>C</i> .
<i>alpha</i>	This parameter scales the input matrix by <i>alpha</i> .
<i>a</i>	Array.
<i>lda</i>	<p>Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix <i>A</i>; measured in the number of elements.</p> <p>For <i>ordering</i> = 'C' or 'c': when <i>transa</i> = 'N', 'n', 'R', or 'r', <i>lda</i> must be at least $\max(1, m)$; otherwise <i>lda</i> must be $\max(1, n)$.</p> <p>For <i>ordering</i> = 'R' or 'r': when <i>transa</i> = 'N', 'n', 'R', or 'r', <i>lda</i> must be at least $\max(1, n)$; otherwise <i>lda</i> must be $\max(1, m)$.</p>
<i>beta</i>	This parameter scales the input matrix by <i>beta</i> .
<i>b</i>	Array.

<i>ldb</i>	<p>Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix <i>B</i>; measured in the number of elements.</p> <p>For <i>ordering</i> = 'C' or 'c': when <i>transa</i> = 'N', 'n', 'R', or 'r', <i>ldb</i> must be at least $\max(1, m)$; otherwise <i>ldb</i> must be $\max(1, n)$.</p> <p>For <i>ordering</i> = 'R' or 'r': when <i>transa</i> = 'N', 'n', 'R', or 'r', <i>ldb</i> must be at least $\max(1, n)$; otherwise <i>ldb</i> must be $\max(1, m)$.</p>
<i>ldc</i>	<p>Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix <i>C</i>; measured in the number of elements.</p> <p>If <i>ordering</i> = 'C' or 'c', then <i>ldc</i> must be at least $\max(1, m)$, otherwise <i>ldc</i> must be at least $\max(1, n)$.</p>

Output Parameters

<i>c</i>	Array.
----------	--------

Interfaces

cblas_?gemm_pack_get_size, cblas_gemm_*_pack_get_size

Returns the number of bytes required to store the packed matrix.

Syntax

```
size_t cblas_hgemm_pack_get_size (const CBLAS_IDENTIFIER identifier, const MKL_INT m,
const MKL_INT n, const MKL_INT k)

size_t cblas_sgemm_pack_get_size (const CBLAS_IDENTIFIER identifier, const MKL_INT m,
const MKL_INT n, const MKL_INT k)

size_t cblas_dgemm_pack_get_size (const CBLAS_IDENTIFIER identifier, const MKL_INT m,
const MKL_INT n, const MKL_INT k)

size_t cblas_gemm_s8u8s32_pack_get_size (const CBLAS_IDENTIFIER identifier, const
MKL_INT m, const MKL_INT n, const MKL_INT k)

size_t cblas_gemm_s16s16s32_pack_get_size (const CBLAS_IDENTIFIER identifier, const
MKL_INT m, const MKL_INT n, const MKL_INT k)

size_t cblas_gemm_bf16bf16f32_pack_get_size (const CBLAS_IDENTIFIER identifier, const
MKL_INT m, const MKL_INT n, const MKL_INT k)

size_t cblas_gemm_f16f16f32_pack_get_size (const CBLAS_IDENTIFIER identifier, const
MKL_INT m, const MKL_INT n, const MKL_INT k)
```

Include Files

- mkl.h

Description

The `cblas_?gemm_pack_get_size` and `cblas_gemm_*_pack_get_size` routines belong to a set of related routines that enable the use of an internal packed storage. Call the `cblas_?gemm_pack_get_size` and `cblas_gemm_*_pack_get_size` routines first to query the size of storage required for a packed matrix structure to be used in subsequent calls. Ultimately, the packed matrix structure is used to compute

$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$ for bfloat16, half, single and double precision or

$C := \alpha * (\text{op}(A) + A_offset) * (\text{op}(B) + B_offset) + \beta * C + C_offset$ for integer type.

where:

$\text{op}(X)$ is one of the operations $\text{op}(X) = X$ or $\text{op}(X) = X^T$

α and β are scalars,

A , A_offset , B , B_offset , C , and C_offset are matrices

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

A_offset is an m -by- k matrix.

B_offset is an k -by- n matrix.

C_offset is an m -by- n matrix.

Input Parameters

Parameter	Type	Description
<i>identifier</i>	CBLAS_IDENTIFIER	Specifies which matrix is to be packed: If <i>identifier</i> = CblasAMatrix, the size returned is the size required to store matrix <i>A</i> in an internal format. If <i>identifier</i> = CblasBMatrix, the size returned is the size required to store matrix <i>B</i> in an internal format.
<i>m</i>	MKL_INT	Specifies the number of rows of matrix $\text{op}(A)$ and of the matrix <i>C</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	MKL_INT	Specifies the number of columns of matrix $\text{op}(B)$ and the number of columns of matrix <i>C</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	MKL_INT	Specifies the number of columns of matrix $\text{op}(A)$ and the number of rows of matrix $\text{op}(B)$. The value of <i>k</i> must be at least zero.

Return Values

Parameter	Type	Description
<i>size</i>	size_t	Returns the size (in bytes) required to store the matrix when packed into the internal format of Intel® oneAPI Math Kernel Library (oneMKL).

Example

See the following examples in the MKL installation directory to understand the use of these routines:

cblas_hgemm_pack_get_size: examples\cblas\source\cblas_hgemm_computex.c

cblas_sgemm_pack_get_size: examples\cblas\source\cblas_sgemm_computex.c

cblas_dgemm_pack_get_size: examples\cblas\source\cblas_dgemm_computex.c

`cblas_gemm_s8u8s32_pack_get_size`: `examples\cblas\source\cblas_gemm_s8u8s32_computex.c`

`cblas_gemm_s16u16s32_pack_get_size`: `examples\cblas\source\cblas_gemm_s16s16s32_computex.c`

`cblas_gemm_bf16bf16f32_pack_get_size`: `examples\cblas\source\cblas_gemm_bf16bf16f32_computex.c`

`cblas_gemm_f16f16f32_pack_get_size`: `examples\cblas\source\cblas_gemm_f16f16f32_computex.c`

See Also

`cblas_?gemm_pack` and `cblas_gemm_*_pack`

to pack the matrix into a buffer allocated previously.

`cblas_?gemm_compute` and `cblas_gemm_*_compute`

to compute a matrix-matrix product with general matrices (where one or both input matrices are stored in a packed data structure) and add the result to a scalar-matrix product.

`cblas_?gemm_pack`

Performs scaling and packing of the matrix into the previously allocated buffer.

Syntax

```
void cblas_hgemm_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER identifier,
const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT k, const
MKL_F16 alpha, const MKL_F16 *src, const MKL_INT ld, MKL_F16 *dest);
```

```
void cblas_sgemm_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER identifier,
const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT k, const
float alpha, const float *src, const MKL_INT ld, float *dest);
```

```
void cblas_dgemm_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER identifier,
const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT k, const
double alpha, const double *src, const MKL_INT ld, double *dest);
```

Include Files

- `mkl.h`

Description

The `cblas_?gemm_pack` routine is one of a set of related routines that enable use of an internal packed storage. Call `cblas_?gemm_pack` after you allocate a buffer whose size is given by `cblas_?gemm_pack_getsize`. The `cblas_?gemm_pack` routine scales the identified matrix by `alpha` and packs it into the buffer allocated previously.

NOTE

Do not copy the packed matrix to a different address because the internal implementation depends on the alignment of internally-stored metadata.

The `cblas_?gemm_pack` routine performs this operation:

$$dest := alpha * op(src) \text{ as part of the computation } C := alpha * op(A) * op(B) + beta * C$$

where:

$op(X)$ is one of the operations $op(X) = X$, $op(X) = X^T$, or $op(X) = X^H$,

$alpha$ and $beta$ are scalars,

src is a matrix,

A , B , and C are matrices

$\text{op}(\text{src})$ is an m -by- k matrix if $\text{identifier} = \text{CblasAMatrix}$,
 $\text{op}(\text{src})$ is a k -by- n matrix if $\text{identifier} = \text{CblasBMatrix}$,
 dest is an internal packed storage buffer.

NOTE

You must use the same value of the *Layout* parameter for the entire sequence of related `cblas_?gemm_pack` and `cblas_?gemm_compute` calls.

For best performance, use the same number of threads for packing and for computing.

If packing for both A and B matrices, you must use the same number of threads for packing A as for packing B .

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>identifier</i>	Specifies which matrix is to be packed: If $\text{identifier} = \text{CblasAMatrix}$, the routine allocates storage to pack matrix A . If $\text{identifier} = \text{CblasBMatrix}$, the routine allocates storage to pack matrix B .
<i>trans</i>	Specifies the form of $\text{op}(\text{src})$ used in the packing: If $\text{trans} = \text{CblasNoTrans}$ $\text{op}(\text{src}) = \text{src}$. If $\text{trans} = \text{CblasTrans}$ $\text{op}(\text{src}) = \text{src}^T$. If $\text{trans} = \text{CblasConjTrans}$ $\text{op}(\text{src}) = \text{src}^H$.
<i>m</i>	Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C . The value of m must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C . The value of n must be at least zero.
<i>k</i>	Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of k must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>src</i>	Array:

	<i>identifier</i> = CblasAMatrix		<i>identifier</i> = CblasBMatrix	
	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans or <i>trans</i> = CblasConjTrans	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans or <i>trans</i> = CblasConjTrans
<i>Layout</i> = CblasColMajor	Size $ld*k$. Before entry, the leading m -by- k part of the array <i>src</i> must contain the matrix <i>A</i> .	Size $ld*m$. Before entry, the leading k -by- m part of the array <i>src</i> must contain the matrix <i>A</i> .	Size $ld*n$. Before entry, the leading k -by- n part of the array <i>src</i> must contain the matrix <i>B</i> .	Size $ld*k$. Before entry, the leading n -by- k part of the array <i>src</i> must contain the matrix <i>B</i> .
<i>Layout</i> = CblasRowMajor	Size $ld*m$. Before entry, the leading k -by- m part of the array <i>src</i> must contain the matrix <i>A</i> .	Size $ld*k$. Before entry, the leading m -by- k part of the array <i>src</i> must contain the matrix <i>A</i> .	Size $ld*k$. Before entry, the leading n -by- k part of the array <i>src</i> must contain the matrix <i>B</i> .	Size $ld*n$. Before entry, the leading k -by- n part of the array <i>src</i> must contain the matrix <i>B</i> .

ld

Specifies the leading dimension of *src* as declared in the calling (sub)program.

	<i>identifier</i> = CblasAMatrix		<i>identifier</i> = CblasBMatrix	
	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans or <i>trans</i> = CblasConjTrans	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans or <i>trans</i> = CblasConjTrans
<i>Layout</i> = CblasColMajor	<i>ld</i> must be at least $\max(1, m)$.	<i>ld</i> must be at least $\max(1, k)$.	<i>ld</i> must be at least $\max(1, k)$.	<i>ld</i> must be at least $\max(1, n)$.
<i>Layout</i> = CblasRowMajor	<i>ld</i> must be at least $\max(1, k)$.	<i>ld</i> must be at least $\max(1, m)$.	<i>ld</i> must be at least $\max(1, n)$.	<i>ld</i> must be at least $\max(1, k)$.

dest

Scaled and packed internal storage buffer.

Output Parameters

dest

Overwritten by the matrix $\alpha * op(src)$.

See Also

[cblas_?gemm_pack_get_size](#) Returns the number of bytes required to store the packed matrix.

[cblas_?gemm_compute](#) Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.

[cblas_?gemm](#)

for a detailed description of general matrix multiplication.

cblas_gemm_*_pack

Pack the matrix into the buffer allocated previously.

Syntax

```
void cblas_gemm_s8u8s32_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER
    identifier, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const
    MKL_INT k, const void *src, const MKL_INT ld, void *dest);
```

```
void cblas_gemm_sl6sl6s32_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER
    identifier, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const
    MKL_INT k, const MKL_INT16 *src, const MKL_INT ld, MKL_INT16 *dest);
```

```
void cblas_gemm_bf16bf16f32_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER
    identifier, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const
    MKL_INT k, const MKL_BF16 *src, const MKL_INT ld, MKL_BF16 *dest);
```

```
void cblas_gemm_f16f16f32_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER
identifier, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const
MKL_INT k, const MKL_F16 *src, const MKL_INT ld, MKL_F16 *dest);
```

Include Files

- mkl.h

Description

The `cblas_gemm_*_pack` routine is one of a set of related routines that enable the use of an internal packed storage. Call `cblas_gemm_*_pack` after you allocate a buffer whose size is given by `cblas_gemm_*_pack_get_size`. The `cblas_gemm_*_pack` routine packs the identified matrix into the buffer allocated previously.

The `cblas_gemm_*_pack` routine performs this operation:

$dest := op(src)$ as part of the computation $C := alpha * (op(A) + A_offset) * (op(B) + B_offset) + beta * C + C_offset$ for integer types.

$C := alpha * op(A) * op(B) + beta * C$ for bfloat16 type.

where:

$op(X)$ is one of the operations $op(X) = X$ or $op(X) = X^T$

$alpha$ and $beta$ are scalars,

src is a matrix,

A , A_offset , B , B_offset , c , and C_offset are matrices

$op(src)$ is an m -by- k matrix if $identifier = CblasAMatrix$,

$op(src)$ is a k -by- n matrix if $identifier = CblasBMatrix$,

$dest$ is the buffer previously allocated to store the matrix packed into an internal format

A_offset is an m -by- k matrix.

B_offset is an k -by- n matrix.

C_offset is an m -by- n matrix.

NOTE

You must use the same value of the `Layout` parameter for the entire sequence of related `cblas_gemm_*_pack` and `cblas_gemm_*_compute` calls.

For best performance, use the same number of threads for packing and for computing.

If packing for both A and B matrices, you must use the same number of threads for packing A as for packing B .

Input Parameters

<code>Layout</code>	CBLAS_LAYOUT Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<code>identifier</code>	CBLAS_IDENTIFIER Specifies which matrix is to be packed: If <code>identifier = CblasAMatrix</code> , the A matrix is packed. If <code>identifier = CblasBMatrix</code> , the B matrix is packed.

*trans***CBLAS_TRANSPOSE**Specifies the form of $\text{op}(src)$ used in the packing:If $trans = \text{CblasNoTrans}$ $\text{op}(src) = src$.If $trans = \text{CblasTrans}$ $\text{op}(src) = src^T$.*m***MKL_INT**Specifies the number of rows of matrix $\text{op}(A)$ and of the matrix C . The value of m must be at least zero.*n***MKL_INT**Specifies the number of columns of matrix $\text{op}(B)$ and the number of columns of matrix C . The value of n must be at least zero.*k***MKL_INT**Specifies the number of columns of matrix $\text{op}(A)$ and the number of rows of matrix $\text{op}(B)$. The value of k must be at least zero.*src*

MKL_BF16* for `cblas_gemm_bf16bf16f32_pack`, MKL_F16* for `cblas_gemm_f16f16f32_pack`, void* for `cblas_gemm_s8u8s32_pack` and MKL_INT16* for `cblas_gemm_s16s16s32_pack`

	<i>identifier</i> = CblasAMatrix		<i>identifier</i> = CblasBMatrix	
	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans
<i>Layout</i> = CblasColMajor	Size $ld*k$. Before entry, the leading m -by- k part of the array <i>src</i> must contain the matrix <i>A</i> . For <code>cblas_gemm_s8u8s32_pack</code> the element in <i>src</i> array must be an 8-bit signed integer.	Size $ld*m$. Before entry, the leading k -by- m part of the array <i>src</i> must contain the matrix <i>A</i> . For <code>cblas_gemm_s8u8s32_pack</code> the element in <i>src</i> array must be an 8-bit signed integer.	Size $ld*n$. Before entry, the leading k -by- n part of the array <i>src</i> must contain the matrix <i>B</i> . For <code>cblas_gemm_s8u8s32_pack</code> the element in <i>src</i> array must be an 8-bit unsigned integer.	Size $ld*k$. Before entry, the leading n -by- k part of the array <i>src</i> must contain the matrix <i>B</i> . For <code>cblas_gemm_s8u8s32_pack</code> the element in <i>src</i> array must be an 8-bit unsigned integer.

	<i>identifier</i> = CblasAMatrix		<i>identifier</i> = CblasBMatrix	
	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans
<i>Layout</i> = CblasRowMajor	<p>Size $ld*m$.</p> <p>Before entry, the leading k-by-m part of the array <i>src</i> must contain the matrix <i>A</i>.</p> <p>For <code>cblas_gemm_s8u8s32_pack</code> the element in <i>src</i> array must be an 8-bit unsigned integer.</p>	<p>Size $ld*k$.</p> <p>Before entry, the leading m-by-k part of the array <i>src</i> must contain the matrix <i>A</i>.</p> <p>For <code>cblas_gemm_s8u8s32_pack</code> the element in <i>src</i> array must be an 8-bit unsigned integer.</p>	<p>Size $ld*k$.</p> <p>Before entry, the leading n-by-k part of the array <i>src</i> must contain the matrix <i>B</i>.</p> <p>For <code>cblas_gemm_s8u8s32_pack</code> the element in <i>src</i> array must be an 8-bit signed integer.</p>	<p>Size $ld*n$.</p> <p>Before entry, the leading k-by-n part of the array <i>src</i> must contain the matrix <i>B</i>.</p> <p>For <code>cblas_gemm_s8u8s32_pack</code> the element in <i>src</i> array must be an 8-bit signed integer.</p>

ld

MKL_INT Specifies the leading dimension of *src* as declared in the calling (sub)program.

	<i>identifier</i> = CblasAMatrix		<i>identifier</i> = CblasBMatrix	
	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans
<i>Layout</i> = CblasColMajor	<i>ld</i> must be at least $\max(1, m)$.	<i>ld</i> must be at least $\max(1, k)$.	<i>ld</i> must be at least $\max(1, k)$.	<i>ld</i> must be at least $\max(1, n)$.

	<i>identifier</i> = CblasAMatrix		<i>identifier</i> = CblasBMatrix	
	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans	<i>trans</i> = CblasNoTrans	<i>trans</i> = CblasTrans
<i>Layout</i> = CblasRowMajor	<i>ld</i> must be at least $\max(1, k)$.	<i>ld</i> must be at least $\max(1, m)$.	<i>ld</i> must be at least $\max(1, n)$.	<i>ld</i> must be at least $\max(1, k)$.

dest

MKL_BF16* for `cblas_gemm_bf16bf16f32_pack`, MKL_F16* for `cblas_gemm_f16f16f32_pack`, `void*` for `cblas_gemm_s8u8s32_pack` or MKL_INT16* for `cblas_gemm_s16s16s32_pack`
Buffer for the packed matrix.

Output Parameters

dest

MKL_BF16* for `cblas_gemm_bf16bf16f32_pack`, MKL_F16* for `cblas_gemm_f16f16f32_pack`, `void*` for `cblas_gemm_s8u8s32_pack` or MKL_INT16* for `cblas_gemm_s16s16s32_pack`

Overwritten by the matrix `op(src)` stored in a format internal to Intel® oneAPI Math Kernel Library (oneMKL).

Example

See the following examples in the MKL installation directory to understand the use of these routines:

`cblas_gemm_s8u8s32_pack`: `examples\cblas\source\cblas_gemm_s8u8s32_computex.c`

`cblas_gemm_s16s16s32_pack`: `examples\cblas\source\cblas_gemm_s16s16s32_computex.c`

`cblas_gemm_bf16bf16f32_pack`: `examples\cblas\source\cblas_gemm_bf16bf16f32_computex.c`

`cblas_gemm_f16f16f32_pack`: `examples\cblas\source\cblas_gemm_f16f16f32_computex.c`

Application Notes

When using `cblas_gemm_s8u8s32_pack` with row-major layout, the data types of *A* and *B* must be swapped. That is, you must provide an 8-bit unsigned integer array for matrix *A* and an 8-bit signed integer array for matrix *B*.

See Also

`cblas_gemm*_pack_get_size`

to return the number of bytes needed to store the packed matrix.

`cblas_gemm*_compute`

to compute a matrix-matrix product with general integer matrices (where one or both input matrices are stored in a packed data structure) and add the result to a scalar-matrix product.

cblas_?gemm_compute

Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.

Syntax

```
void cblas_hgemm_compute (const CBLAS_LAYOUT Layout, const MKL_INT transa, const
MKL_INT transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const MKL_F16 *a,
const MKL_INT lda, const MKL_F16 *b, const MKL_INT ldb, const MKL_F16 beta, MKL_F16 *c,
const MKL_INT ldc);
```

```
void cblas_sgemm_compute (const CBLAS_LAYOUT Layout, const MKL_INT transa, const
MKL_INT transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const float *a,
const MKL_INT lda, const float *b, const MKL_INT ldb, const float beta, float *c, const
MKL_INT ldc);
```

```
void cblas_dgemm_compute (const CBLAS_LAYOUT Layout, const MKL_INT transa, const
MKL_INT transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const double *a,
const MKL_INT lda, const double *b, const MKL_INT ldb, const double beta, double *c,
const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The `cblas_?gemm_compute` routine is one of a set of related routines that enable use of an internal packed storage. After calling `cblas_?gemm_pack` call `cblas_?gemm_compute` to compute

$$C := \text{op}(A) * \text{op}(B) + \text{beta} * C,$$

where:

- $\text{op}(X)$ is one of the operations $\text{op}(X) = X$, $\text{op}(X) = X^T$, or $\text{op}(X) = X^H$,
- beta is a scalar,
- A , B , and C are matrices:
- $\text{op}(A)$ is an m -by- k matrix,
- $\text{op}(B)$ is a k -by- n matrix,
- C is an m -by- n matrix.

NOTE

You must use the same value of the `Layout` parameter for the entire sequence of related `cblas_?gemm_pack` and `cblas_?gemm_compute` calls.

For best performance, use the same number of threads for packing and for computing.

If packing for both A and B matrices, you must use the same number of threads for packing A as for packing B .

Input Parameters

<code>Layout</code>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
---------------------	---

transa

Specifies the form of $\text{op}(A)$ used in the matrix multiplication, one of the CBLAS_TRANSPOSE or CBLAS_STORAGE enumerated types:

If *transa* = CblasNoTrans $\text{op}(A) = A$.

If *transa* = CblasTrans $\text{op}(A) = A^T$.

If *transa* = CblasConjTrans $\text{op}(A) = A^H$.

If *transa* = CblasPacked the matrix in array *a* is packed and *lda* is ignored.

transb

Specifies the form of $\text{op}(B)$ used in the matrix multiplication, one of the CBLAS_TRANSPOSE or CBLAS_STORAGE enumerated types:

If *transb* = CblasNoTrans $\text{op}(B) = B$.

If *transb* = CblasTrans $\text{op}(B) = B^T$.

If *transb* = CblasConjTrans $\text{op}(B) = B^H$.

If *transb* = CblasPacked the matrix in array *b* is packed and *ldb* is ignored.

m

Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix *C*. The value of *m* must be at least zero.

n

Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix *C*. The value of *n* must be at least zero.

k

Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of *k* must be at least zero.

a

Array:

	<i>transa</i> = CblasNoTrans	<i>transa</i> = CblasTrans or <i>transa</i> = CblasConjTrans	<i>transa</i> = CblasPacked
<i>Layout</i> = CblasColMajor	Size <i>lda</i> * <i>k</i> . Before entry, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Size <i>lda</i> * <i>m</i> . Before entry, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Stored in internal packed format.
<i>Layout</i> = CblasRowMajor	Size <i>lda</i> * <i>m</i> . Before entry, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Size <i>lda</i> * <i>k</i> . Before entry, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Stored in internal packed format.

lda

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

	<i>transa</i> = CblasNoTrans	<i>transa</i> = CblasTrans or <i>transa</i> = CblasConjTrans	<i>transa</i> = CblasPacked
<i>Layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1, m)$.	<i>lda</i> must be at least $\max(1, k)$.	<i>lda</i> is ignored.
<i>Layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1, k)$.	<i>lda</i> must be at least $\max(1, m)$.	<i>lda</i> is ignored.

b

Array:

	<i>transb</i> = CblasNoTrans	<i>transb</i> = CblasTrans or <i>transb</i> = CblasConjTrans	<i>transb</i> = CblasPacked
<i>Layout</i> = CblasColMajor	Size $ldb*n$. Before entry, the leading k -by- n part of the array <i>b</i> must contain the matrix <i>B</i> .	Size $ldb*k$. Before entry, the leading n -by- k part of the array <i>b</i> must contain the matrix <i>B</i> .	Stored in internal packed format.
<i>Layout</i> = CblasRowMajor	Size $ldb*k$. Before entry, the leading n -by- k part of the array <i>b</i> must contain the matrix <i>B</i> .	Size $ldb*n$. Before entry, the leading k -by- n part of the array <i>b</i> must contain the matrix <i>B</i> .	Stored in internal packed format.

*ldb*Specifies the leading dimension of *b* as declared in the calling (sub)program.

	<i>transb</i> = CblasNoTrans	<i>transb</i> = CblasTrans or <i>transb</i> = CblasConjTrans	<i>transb</i> = CblasPacked
<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> is ignored.
<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> is ignored.

beta

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

c

Array:

<i>Layout</i> = CblasColMajor	<p>Size <i>ldc</i>*<i>n</i>.</p> <p>Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i>, except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.</p>
<i>Layout</i> = CblasRowMajor	<p>Size <i>ldc</i>*<i>m</i>.</p> <p>Before entry, the leading <i>n</i>-by-<i>m</i> part of the array <i>c</i> must contain the matrix <i>C</i>, except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.</p>

ldc

Specifies the leading dimension of *c* as declared in the calling (sub)program.

<i>Layout</i> = CblasColMajor	<i>ldc</i> must be at least $\max(1, m)$.
<i>Layout</i> = CblasRowMajor	<i>ldc</i> must be at least $\max(1, n)$.

Output Parameters

c

Overwritten by the *m*-by-*n* matrix $\text{op}(A) * \text{op}(B) + \text{beta} * C$.

See Also

[cblas_?gemm_pack_get_size](#) Returns the number of bytes required to store the packed matrix.

[cblas_?gemm_pack](#) Performs scaling and packing of the matrix into the previously allocated buffer.

[cblas_?gemm](#)
for a detailed description of general matrix multiplication.

cblas_gemm_*_compute

Computes a matrix-matrix product with general integer matrices (where one or both input matrices are stored in a packed data structure) and adds the result to a scalar-matrix product.

Syntax

```
void cblas_gemm_s8u8s32_compute(const CBLAS_LAYOUT Layout, const MKL_INT transa, const
MKL_INT transb, const CBLAS_OFFSET offsetc, const MKL_INT m, const MKL_INT n, const
MKL_INT k, const float alpha, const void *a, const MKL_INT lda, const MKL_INT8 oa,
const void *b, const MKL_INT ldb, const MKL_INT8 ob, const float beta, MKL_INT32 *c,
const MKL_INT ldc, const MKL_INT32 *oc);
```

```
void cblas_gemm_s16s16s32_compute(const CBLAS_LAYOUT Layout, const MKL_INT transa,
const MKL_INT transb, const CBLAS_OFFSET offsetc, const MKL_INT m, const MKL_INT n,
const MKL_INT k, const float alpha, const MKL_INT16 *a, const MKL_INT lda, const
MKL_INT16 oa, const MKL_INT16 *b, const MKL_INT ldb, const MKL_INT16 ob, const float
beta, MKL_INT32 *c, const MKL_INT ldc, const MKL_INT32 *oc);
```

Include Files

- `mkl.h`

Description

The `cblas_gemm_*_compute` routine is one of a set of related routines that enable use of an internal packed storage. After calling `cblas_gemm_*_pack` call `cblas_gemm_*_compute` to compute

$$C := \alpha \cdot (\text{op}(A) + A_offset) \cdot (\text{op}(B) + B_offset) + \beta \cdot C + C_offset,$$

where:

$\text{op}(X)$ is either $\text{op}(X) = X$ or $\text{op}(X) = X^T$

α and β are scalars

A , B , and C are matrices:

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

A_offset is an m -by- k matrix with every element equal to the value `oa`.

B_offset is an k -by- n matrix with every element equal to the value `ob`.

C_offset is an m -by- n matrix defined by the `oc` array as described in the description of the `offsetc` parameter.

NOTE

You must use the same value of the `Layout` parameter for the entire sequence of related `cblas_?gemm_pack` and `cblas_?gemm_compute` calls.

For best performance, use the same number of threads for packing and for computing.

If you are packing for both A and B matrices, you must use the same number of threads for packing A as for packing B .

Input Parameters

<code>Layout</code>	CBLAS_LAYOUT Specifies whether two-dimensional array storage is row-major (<code>CblasRowMajor</code>) or column-major (<code>CblasColMajor</code>).
<code>transa</code>	MKL_INT Specifies the form of $\text{op}(A)$ used in the packing: If <code>transa = CblasNoTrans</code> $\text{op}(A) = A$. If <code>transa = CblasTrans</code> $\text{op}(A) = A^T$. If <code>transa = CblasPacked</code> the matrix in array <code>a</code> is packed into a format internal to Intel® oneAPI Math Kernel Library (oneMKL) and <code>lda</code> is ignored.
<code>transb</code>	MKL_INT Specifies the form of $\text{op}(B)$ used in the packing: If <code>transb = CblasNoTrans</code> $\text{op}(B) = B$. If <code>transb = CblasTrans</code> $\text{op}(B) = B^T$. If <code>transb = CblasPacked</code> the matrix in array <code>b</code> is packed into a format internal to Intel® oneAPI Math Kernel Library (oneMKL) and <code>ldb</code> is ignored.
<code>offsetc</code>	CBLAS_OFFSET Specifies the form of C_offset used in the matrix multiplication.

If `offsetc=CblasFixOffset` :`oc` has a single element and every element of `C_offset` is equal to this element.

If `offsetc=CblasColOffset` :`oc` has a size of m and every element of `C_offset` is equal to `oc`.

If `offsetc=CblasRowOffset` :`oc` has a size of n and every element of `C_offset` is equal to `oc`.

m MKL_INT Specifies the number of rows of the matrix $op(A)$ and of the matrix C . The value of m must be at least zero.

n MKL_INT Specifies the number of columns of the matrix $op(B)$ and the number of columns of the matrix C . The value of n must be at least zero.

k MKL_INT Specifies the number of columns of the matrix $op(A)$ and the number of rows of the matrix $op(B)$. The value of k must be at least zero.

α float Specifies the scalar α .

a void* for `gemm_s8u8s32_compute`

MKL_INT16* for `gemm_s16s16s32_compute`

Layout = CblasColMajor	
<code>transa = CblasNoTrans</code>	<p>Array, size $lda*k$.</p> <p>Before entry, the leading m-by-k part of the array a must contain the matrix A.</p> <p>For <code>cblas_gemm_s8u8s32_compute</code>, the element in the a array must be an 8-bit signed integer.</p>
<code>transa = CblasTrans</code>	<p>Array, size $lda*m$.</p> <p>Before entry, the leading k-by-m part of the array a must contain the matrix A.</p> <p>For <code>cblas_gemm_s8u8s32_compute</code>, the element in the a array must be an 8-bit signed integer.</p>
<code>transa = CblasPacked</code>	<p>Array of size returned by <code>cblas_gemm_*_pack_get_size</code> and initialized using <code>cblas_gemm_*_pack</code></p>
Layout = CblasRowMajor	
<code>transa = CblasNoTrans</code>	<p>Array, size $lda*m$.</p> <p>Before entry, the leading k-by-m part of the array a must contain the matrix A.</p> <p>For <code>cblas_gemm_s8u8s32_compute</code>, the element in the a array must be an 8-bit unsigned integer.</p>
<code>transa = CblasTrans</code>	<p>Array, size $lda*k$.</p>

<i>Layout</i> = CblasRowMajor	
	Before entry, the leading m -by- k part of the array a must contain the matrix A .
	For <code>cblas_gemm_s8u8s32_compute</code> , the element in the a array must be an 8-bit unsigned integer.
<code>transa = CblasPacked</code>	Array size returned by <code>cblas_gemm_*_pack_get_size</code> and initialized using <code>cblas_gemm_*_pack</code>

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

	<i>transa</i> = CblasNoTrans	<i>transa</i> = CblasTrans
<i>Layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1, m)$.	<i>lda</i> must be at least $\max(1, k)$.
<i>Layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1, k)$.	<i>lda</i> must be at least $\max(1, m)$.

oa MKL_INT8 for `cblas_gemm_s8u8s32_compute`

MKL_INT16 for `cblas_gemm_s16s16s32_compute`

Specifies the scalar offset value for the matrix A .

b void* for `gemm_s8u8s32_compute`

MKL_INT16* for `gemm_s16s16s32_compute`

<i>Layout</i> = CblasColMajor	
<i>transa</i> = CblasNoTrans	Array, size $ldb*n$. Before entry, the leading k -by- n part of the array b must contain the matrix B . For <code>cblas_gemm_s8u8s32_compute</code> , the element in the b array must be an 8-bit unsigned integer.
<i>transa</i> = CblasTrans	Array, size $ldb*k$. Before entry, the leading n -by- k part of the array b must contain the matrix B . For <code>cblas_gemm_s8u8s32_compute</code> , the element in the b array must be an 8-bit unsigned integer.
<i>transa</i> = CblasPacked	Array of size returned by <code>cblas_gemm_*_pack_get_size</code> and initialized using <code>cblas_gemm_*_pack</code>

<i>Layout</i> = CblasRowMajor	
<i>transa</i> = CblasNoTrans	<p>Array, size $ldb*k$.</p> <p>Before entry, the leading n-by-k part of the array <i>b</i> must contain the matrix <i>B</i>.</p> <p>For <code>cblas_gemm_s8u8s32_compute</code>, the element in the <i>b</i> array must be an 8-bit signed integer.</p>
<i>transa</i> = CblasTrans	<p>Array, size $ldb*n$.</p> <p>Before entry, the leading k-by-n part of the array <i>b</i> must contain the matrix <i>B</i>.</p> <p>For <code>cblas_gemm_s8u8s32_compute</code>, the element in the <i>b</i> array must be an 8-bit signed integer.</p>
<i>transa</i> = CblasPacked	<p>Array of size returned by <code>cblas_gemm*_pack_get_size</code> and initialized using <code>cblas_gemm*_pack</code></p>

*ldb*MKL_INT Specifies the leading dimension of *b* as declared in the calling (sub)program.

	<i>transb</i> = CblasNoTrans	<i>transb</i> = CblasTrans
<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> must be at least $\max(1, n)$.
<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$.

*ob*MKL_INT8 for `cblas_gemm_s8u8s32_compute`MKL_INT16 for `cblas_gemm_s16s16s32_compute`Specifies the scalar offset value for the matrix *B*.*beta*

float

Specifies the scalar *beta*.*c*

MKL_INT32*

Array:

<i>Layout</i> = CblasColMajor	<p>Array, size $ldc*n$.</p> <p>Before entry, the leading m-by-n part of the array <i>c</i> must contain the matrix <i>C</i>, except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.</p>
<i>Layout</i> = CblasRowMajor	Array, size $ldc*m$.

	Before entry, the leading n -by- m part of the array c must contain the matrix C , except when β is equal to zero, in which case c need not be set on entry.
--	--

ldc MKL_INT Specifies the leading dimension of c as declared in the calling (sub)program.

<i>Layout</i> = CblasColMajor	<i>ldc</i> must be at least $\max(1, m)$
<i>Layout</i> = CblasRowMajor	<i>ldc</i> must be at least $\max(1, n)$

oc MKL_INT32*

Array, size *len*. Specifies the scalar offset value for the matrix C .

If *offsetc* = CblasFixOffset, *len* must be at least 1.

If *offsetc* = CblasColOffset, *len* must be at least $\max(1, m)$.

If *offsetc* = CblasRowOffset, *len* must be at least $\max(1, n)$.

Output Parameters

c MKL_INT32*
Overwritten by the matrix $\alpha * (\text{op}(A) + A_offset) * (\text{op}(B) + B_offset) + \beta * C + C_offset$.

Example

See the following examples in the MKL installation directory to understand the use of these routines:

`cblas_gemm_s8u8s32_compute`: `examples\cblas\source\cblas_gemm_s8u8s32_computex.c`

`cblas_gemm_s16s16s32_compute`: `examples\cblas\source\cblas_gemm_s16s16s32_computex.c`

Application Notes

You can expand the matrix-matrix product in this manner:

$$(\text{op}(A) + A_offset) * (\text{op}(B) + B_offset) = \text{op}(A) * \text{op}(B) + \text{op}(A) * B_offset + A_offset * \text{op}(B) + A_offset * B_offset$$

After computing these four multiplication terms separately, they are summed from left to right. The results from the matrix-matrix product and the C matrix are scaled with α and β floating-point values respectively using double-precision arithmetic. Before storing the results to the output c array, the floating-point values are rounded to the nearest integers.

In the event of overflow or underflow, the results depend on the architecture. The results are either unsaturated (wrapped) or saturated to maximum or minimum representable integer values for the data type of the output matrix.

When using `cblas_gemm_s8u8s32_compute` with row-major layout, the data types of A and B must be swapped. That is, you must provide an 8-bit unsigned integer array for matrix A and an 8-bit signed integer array for matrix B .

See Also

`cblas_gemm_*_pack_get_size`

to return the number of bytes needed to store the packed matrix.

`cblas_gemm_*_pack`

to pack the matrix into the buffer allocated previously.

cblas_gemm_bf16bf16f32_compute

Computes a matrix-matrix product with general bfloat16 matrices (where one or both input matrices are stored in a packed data structure) and adds the result to a scalar-matrix product.

Syntax

C:

```
void cblas_gemm_bf16bf16f32_compute (const CBLAS_LAYOUT Layout, const MKL_INT transa,
const MKL_INT transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const float
alpha, const MKL_BF16 *a, const MKL_INT lda, const MKL_BF16 *b, const MKL_INT ldb,
const float beta, float *c, const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The `cblas_gemm_bf16bf16f32_compute` routine is one of a set of related routines that enable use of an internal packed storage. After calling `cblas_gemm_bf16bf16f32_pack` call `cblas_gemm_bf16bf16f32_compute` to compute

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C,$$

where:

- $\text{op}(X)$ is either $\text{op}(X) = X$ or $\text{op}(X) = X^T$,
- α and β are scalars,
- A , B , and C are matrices:
- $\text{op}(A)$ is an m -by- k matrix,
- $\text{op}(B)$ is a k -by- n matrix,
- C is an m -by- n matrix.

NOTE

You must use the same value of the `Layout` parameter for the entire sequence of related `cblas_gemm_bf16bf16f32_pack` and `cblas_gemm_bf16bf16f32_compute` calls.

For best performance, use the same number of threads for packing and for computing.

If packing for both A and B matrices, you must use the same number of threads for packing A as for packing B .

Input Parameters

<code>Layout</code>	CBLAS_LAYOUT Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<code>transa</code>	MKL_INT Specifies the form of $\text{op}(A)$ used in the packing: If <code>transa = CblasNoTrans</code> $\text{op}(A) = A$.

If $transa = \text{CblasTrans}$ $\text{op}(A) = A^T$.

If $transa = \text{CblasPacked}$ the matrix in array a is packed into a format internal to Intel® oneAPI Math Kernel Library (oneMKL) and lda is ignored.

$transb$

MKL_INT

Specifies the form of $\text{op}(B)$ used in the packing:

If $transb = \text{CblasNoTrans}$ $\text{op}(B) = B$.

If $transb = \text{CblasTrans}$ $\text{op}(B) = B^T$.

If $transb = \text{CblasPacked}$ the matrix in array b is packed into a format internal to Intel® oneAPI Math Kernel Library (oneMKL) and ldb is ignored.

m

MKL_INT

Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C . The value of m must be at least zero.

n

MKL_INT

Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C . The value of n must be at least zero.

k

MKL_INT

Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of k must be at least zero.

$alpha$

float

Specifies the scalar $alpha$.

a

MKL_BF16*

	$transa = \text{CblasNoTrans}$	$transa = \text{CblasTrans}$	$transa = \text{CblasPacked}$
$Layout = \text{CblasColMajor}$	Array, size $lda*k$. Before entry, the leading m -by- k part of the array a must contain the matrix A .	Array, size $lda*m$. Before entry, the leading k -by- m part of the array a must contain the matrix A .	Array of size returned by <code>cblas_gemm_bf16bf16f32_pac</code> and initialized using <code>cblas_gemm_bf16bf16f32_pac</code>
$Layout = \text{CblasRowMajor}$	Array, size $lda*m$. Before entry, the leading k -by- m part of	Array, size $lda*k$. Before entry, the leading m -	Array size returned by <code>cblas_gemm_bf16bf16f32_pac</code> and initialized using <code>cblas_gemm_bf16bf16f32_pac</code>

	the array <i>a</i> must contain the matrix <i>A</i> .	by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	
--	---	--	--

lda

MKL_INT

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

	<i>transa</i> = CblasNoTrans	<i>transa</i> = CblasTrans
<i>Layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1, m)$.	<i>lda</i> must be at least $\max(1, k)$.
<i>Layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1, k)$.	<i>lda</i> must be at least $\max(1, m)$.

b

MKL_BF16*

	<i>transa</i> = CblasNoTrans	<i>transa</i> = CblasTrans	<i>transa</i> = CblasPacked
<i>Layout</i> = CblasColMajor	Array, size $ldb*n$. Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size $ldb*k$. Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array of size returned by <code>cblas_gemm_bf16bf16f32_pac</code> and initialized using <code>cblas_gemm_bf16bf16f32_pac</code>
<i>Layout</i> = CblasRowMajor	Array, size $ldb*k$. Before entry, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size $ldb*n$. Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array size returned by <code>cblas_gemm_bf16bf16f32_pac</code> and initialized using <code>cblas_gemm_bf16bf16f32_pac</code>

ldb

MKL_INT

Specifies the leading dimension of *b* as declared in the calling (sub)program.

	<i>transb</i> = CblasNoTrans	<i>transb</i> = CblasTrans
<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> must be at least $\max(1, n)$.
<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$.

beta

float

Specifies the scalar *beta*.*c*

float*

<i>Layout</i> = CblasColMajor	Array, size $ldb \times n$. Before entry, the leading m -by- n part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.
<i>Layout</i> = CblasRowMajor	Array, size $ldb \times m$. Before entry, the leading n -by- m part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.

ldb

MKL_INT

Specifies the leading dimension of *c* as declared in the calling (sub)program.

<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, m)$.
<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, n)$.

Output Parameters

c

float*

Overwritten by the matrix $\alpha * \text{op}(A) * \text{op}(B) + \text{beta} * C$.

Example

See the following examples in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory to understand the use of these routines:

```
cblas_gemm_bf16bf16f32_compute:
examples\cblas\source\cblas_gemm_bf16bf16f32_computex.c
```

Application Notes

On architectures without native `bfloat16` hardware instructions, matrix *A* and *B* are upconverted to single precision and `SGEMM` is called to compute matrix multiplication operation.

`cblas_gemm_bf16bf16f32`

Computes a matrix-matrix product with general bfloat16 matrices.

Syntax

```
void cblas_gemm_bf16bf16f32 (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa,
const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const
float alpha, const MKL_BF16 *a, const MKL_INT lda, const MKL_BF16 *b, const MKL_INT
ldb, const float beta, float *c, const MKL_INT ldc);
```

Include Files

- `mkl.h`

Description

The `cblas_gemm_bf16bf16f32` routines compute a scalar-matrix-matrix product and adds the result to a scalar-matrix product. The operation is defined as:

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$$

where :

- $\text{op}(X)$ is one of $\text{op}(X) = X$ or $\text{op}(X) = X^T$,
- α and β are scalars,
- A , B , and C are matrices
- $\text{op}(A)$ is m -by- k matrix,
- $\text{op}(B)$ is k -by- n matrix,
- C is an m -by- n matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (<code>CblasRowMajor</code>) or column-major (<code>CblasColMajor</code>).
<i>transa</i>	Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if $\text{transa}=\text{CblasNoTrans}$, then $\text{op}(A) = A$; if $\text{transa}=\text{CblasTrans}$, then $\text{op}(A) = A^T$.
<i>transb</i>	Specifies the form of $\text{op}(B)$ used in the matrix multiplication: if $\text{transb}=\text{CblasNoTrans}$, then $\text{op}(B) = B$; if $\text{transb}=\text{CblasTrans}$, then $\text{op}(B) = B^T$.
<i>m</i>	Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C . The value of m must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C . The value of n must be at least zero.

k Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of *k* must be at least zero.

alpha Specifies the scalar *alpha*.

a

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans
<i>Layout</i> = CblasColMajor	Array, size $lda*k$ Before entry, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size $lda*m$ Before entry, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>Layout</i> = CblasRowMajor	Array, size $lda*m$ Before entry, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix.	Array, size $lda*k$ Before entry, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix.

lda

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

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans
<i>Layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1, m)$.	<i>lda</i> must be at least $\max(1, k)$.
<i>Layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1, k)$.	<i>lda</i> must be at least $\max(1, m)$.

b

	<i>transb</i> =CblasNoTrans	<i>transb</i> =CblasTrans
<i>Layout</i> = CblasColMajor	Array, size <i>ldb</i> by <i>n</i> Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>ldb</i> by <i>k</i> Before entry the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>Layout</i> = CblasRowMajor	Array, size <i>ldb</i> by <i>k</i> Before entry the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>ldb</i> by <i>n</i> Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .

ldb

Specifies the leading dimension of *b* as declared in the calling (sub)program.

	<i>transb</i> =CblasNoTrans	<i>transb</i> =CblasTrans
--	-----------------------------	---------------------------

<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> must be at least $\max(1, n)$.
<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$.

beta

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

c

<i>Layout</i> = CblasColMajor	Array, size <i>ldc</i> by <i>n</i> . Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.
<i>Layout</i> = CblasRowMajor	Array, size <i>ldc</i> by <i>m</i> . Before entry, the leading <i>n</i> -by- <i>m</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.

ldc

Specifies the leading dimension of *c* as declared in the calling (sub)program.

<i>Layout</i> = CblasColMajor	<i>ldc</i> must be at least $\max(1, m)$.
<i>Layout</i> = CblasRowMajor	<i>ldc</i> must be at least $\max(1, n)$.

Output Parameters

c

Overwritten by $\alpha * \text{op}(A) * \text{op}(B) + \text{beta} * C$.

Example

For examples of routine usage, see these code examples in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- `cblas_gemm_bf16bf16f32: examples\cblas\source\cblas_gemm_bf16bf16f32x.c`

Application Notes

On architectures without native bfloat16 hardware instructions, matrix *A* and *B* are upconverted to single precision and `SGEMM` is called to compute matrix multiplication operation.

`cblas_gemm_f16f16f32_compute`

Computes a matrix-matrix product with general matrices of half-precision data type (where one or both input matrices are stored in a packed data structure) and adds the result to a scalar-matrix product.

Syntax

C:

```
void cblas_gemm_f16f16f32_compute (const CBLAS_LAYOUT Layout, const MKL_INT transa,
const MKL_INT transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const float
alpha, const MKL_F16 *a, const MKL_INT lda, const MKL_F16 *b, const MKL_INT ldb, const
float beta, float *C, const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The `cblas_gemm_f16f16f32_compute` routine is one of a set of related routines that enable use of an internal packed storage. After calling `cblas_gemm_f16f16f32_pack` call `cblas_gemm_f16f16f32_compute` to compute

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C,$$

where:

$\text{op}(X)$ is either $\text{op}(X) = X$ or $\text{op}(X) = X^T$,

α and β are scalars,

A , B , and C are matrices:

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

NOTE

You must use the same value of the `Layout` parameter for the entire sequence of related `cblas_gemm_f16f16f32_pack` and `cblas_gemm_f16f16f32_compute` calls.

For best performance, use the same number of threads for packing and for computing.

If packing for both A and B matrices, you must use the same number of threads for packing A as for packing B .

Input Parameters

<code>Layout</code>	CBLAS_LAYOUT Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<code>transa</code>	MKL_INT Specifies the form of $\text{op}(A)$ used in the packing: If <code>transa = CblasNoTrans</code> $\text{op}(A) = A$. If <code>transa = CblasTrans</code> $\text{op}(A) = A^T$. If <code>transa = CblasPacked</code> the matrix in array <code>a</code> is packed into a format internal to Intel® oneAPI Math Kernel Library (oneMKL) and <code>lda</code> is ignored.
<code>transb</code>	MKL_INT Specifies the form of $\text{op}(B)$ used in the packing: If <code>transb = CblasNoTrans</code> $\text{op}(B) = B$.

If $transb = \text{CblasTrans}$ $\text{op}(B) = B^T$.

If $transb = \text{CblasPacked}$ the matrix in array b is packed into a format internal to Intel® oneAPI Math Kernel Library (oneMKL) and ldb is ignored.

m

MKL_INT

Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C . The value of m must be at least zero.

n

MKL_INT

Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C . The value of n must be at least zero.

k

MKL_INT

Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of k must be at least zero.

α

float

Specifies the scalar α .

a

MKL_F16*

	$transa = \text{CblasNoTrans}$	$transa = \text{CblasTrans}$	$transa = \text{CblasPacked}$
$Layout = \text{CblasColMajor}$	Array, size $lda*k$. Before entry, the leading m -by- k part of the array a must contain the matrix A .	Array, size $lda*m$. Before entry, the leading k -by- m part of the array a must contain the matrix A .	Array of size returned by <code>cblas_gemm_f16f16f32_pack</code> and initialized using <code>cblas_gemm_f16f16f32_pack</code> .
$Layout = \text{CblasRowMajor}$	Array, size $lda*m$. Before entry, the leading k -by- m part of the array a must contain the matrix A .	Array, size $lda*k$. Before entry, the leading m -by- k part of the array a must contain the matrix A .	Array size returned by <code>cblas_gemm_f16f16f32_pack</code> and initialized using <code>cblas_gemm_f16f16f32_pack</code> .

lda

MKL_INT

Specifies the leading dimension of a as declared in the calling (sub)program.

	<i>transa</i> = CblasNoTrans	<i>transa</i> = CblasTrans
<i>Layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1, m)$.	<i>lda</i> must be at least $\max(1, k)$.
<i>Layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1, k)$.	<i>lda</i> must be at least $\max(1, m)$.

b

MKL_F16*

	<i>transa</i> = CblasNoTrans	<i>transa</i> = CblasTrans	<i>transa</i> = CblasPacked
<i>Layout</i> = CblasColMajor	Array, size <i>ldb</i> * <i>n</i> . Before entry, the leading <i>k</i> - by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>ldb</i> * <i>k</i> . Before entry, the leading <i>n</i> - by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array of size returned by <code>cblas_gemm_f16f16f32_pack_</code> and initialized using <code>cblas_gemm_f16f16f32_pack</code> .
<i>Layout</i> = CblasRowMajor	Array, size <i>ldb</i> * <i>k</i> . Before entry, the leading <i>n</i> - by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array, size <i>ldb</i> * <i>n</i> . Before entry, the leading <i>k</i> - by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .	Array size returned by <code>cblas_gemm_f16f16f32_pack_</code> and initialized using <code>cblas_gemm_f16f16f32_pack</code> .

ldb

MKL_INT

Specifies the leading dimension of *b* as declared in the calling (sub)program.

	<i>transb</i> = CblasNoTrans	<i>transb</i> = CblasTrans
<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> must be at least $\max(1, n)$.
<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$.

beta

float

Specifies the scalar *beta*.

c

float*

<i>Layout</i> = CblasColMajor	Array, size <i>ldc</i> * <i>n</i> . Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.
<i>Layout</i> = CblasRowMajor	Array, size <i>ldc</i> * <i>m</i> . Before entry, the leading <i>n</i> -by- <i>m</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.

ldc

MKL_INT

Specifies the leading dimension of *c* as declared in the calling (sub)program.

<i>Layout</i> = CblasColMajor	<i>ldc</i> must be at least $\max(1, m)$.
<i>Layout</i> = CblasRowMajor	<i>ldc</i> must be at least $\max(1, n)$.

Output Parameters

c

float*

Overwritten by the matrix $\alpha * \text{op}(A) * \text{op}(B) + \text{beta} * C$.

Example

See the following examples in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory to understand the use of these routines:

```
cblas_gemm_f16f16f32_compute:
examples\cblas\source\cblas_gemm_f16f16f32_computex.c
```

Application Notes

On architectures without native half precision hardware instructions, matrix *A* and *B* are upconverted to single precision and SGEMM is called to compute matrix multiplication operation.

cblas_gemm_f16f16f32

Computes a matrix-matrix product with general matrices of half precision data type.

Syntax

```
void cblas_gemm_f16f16f32 (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa,
const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const
float alpha, const MKL_F16 *a, const MKL_INT lda, const MKL_F16 *b, const MKL_INT ldb,
const float beta, float *c, const MKL_INT ldc);
```

Include Files

- `mkl.h`

Description

The `cblas_gemm_f16f16f32` routines compute a scalar-matrix-matrix product and adds the result to a scalar-matrix product. The operation is defined as:

$$C := \alpha * \text{op}(A) * \text{op}(B) + \beta * C$$

where :

$\text{op}(X)$ is one of $\text{op}(X) = X$ or $\text{op}(X) = X^T$,

α and β are scalars,

A , B , and C are matrices

$\text{op}(A)$ is m -by- k matrix,

$\text{op}(B)$ is k -by- n matrix,

C is an m -by- n matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>transa</i>	Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if $\text{transa}=\text{CblasNoTrans}$, then $\text{op}(A) = A$; if $\text{transa}=\text{CblasTrans}$, then $\text{op}(A) = A^T$.
<i>transb</i>	Specifies the form of $\text{op}(B)$ used in the matrix multiplication: if $\text{transb}=\text{CblasNoTrans}$, then $\text{op}(B) = B$; if $\text{transb}=\text{CblasTrans}$, then $\text{op}(B) = B^T$.
<i>m</i>	Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C . The value of m must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C . The value of n must be at least zero.
<i>k</i>	Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of k must be at least zero.
<i>alpha</i>	Specifies the scalar α .

a

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans
<i>Layout</i> = CblasColMajor	Array, size $lda*k$ Before entry, the leading m -by- k part of the array <i>a</i> must contain the matrix <i>A</i> .	Array, size $lda*m$ Before entry, the leading k -by- m part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>Layout</i> = CblasRowMajor	Array, size $lda*m$	Array, size $lda*k$

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

 lda

Specifies the leading dimension of a as declared in the calling (sub)program.

	$transa=CblasNoTrans$	$transa=CblasTrans$
$Layout = CblasColMajor$	lda must be at least $\max(1, m)$.	lda must be at least $\max(1, k)$.
$Layout = CblasRowMajor$	lda must be at least $\max(1, k)$.	lda must be at least $\max(1, m)$.

 b

	$transb=CblasNoTrans$	$transb=CblasTrans$
$Layout = CblasColMajor$	Array, size ldb by n Before entry, the leading k -by- n part of the array b must contain the matrix B .	Array, size ldb by k Before entry the leading n -by- k part of the array b must contain the matrix B .
$Layout = CblasRowMajor$	Array, size ldb by k Before entry the leading n -by- k part of the array b must contain the matrix B .	Array, size ldb by n Before entry, the leading k -by- n part of the array b must contain the matrix B .

 ldb

Specifies the leading dimension of b as declared in the calling (sub)program.

	$transb=CblasNoTrans$	$transb=CblasTrans$
$Layout = CblasColMajor$	ldb must be at least $\max(1, k)$.	ldb must be at least $\max(1, n)$.
$Layout = CblasRowMajor$	ldb must be at least $\max(1, n)$.	ldb must be at least $\max(1, k)$.

 $beta$

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

 c

$Layout = CblasColMajor$	Array, size ldc by n . Before entry, the leading m -by- n part of the array c must contain the matrix C , except when $beta$ is equal to zero, in which case c need not be set on entry.
--------------------------	--

<i>Layout</i> = CblasRowMajor	Array, size <i>ldc</i> by <i>m</i> . Before entry, the leading <i>n</i> -by- <i>m</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.
----------------------------------	--

ldc Specifies the leading dimension of *c* as declared in the calling (sub)program.

<i>Layout</i> = CblasColMajor	<i>ldc</i> must be at least $\max(1, m)$.
<i>Layout</i> = CblasRowMajor	<i>ldc</i> must be at least $\max(1, n)$.

Output Parameters

c Overwritten by $\alpha * \text{op}(A) * \text{op}(B) + \beta * C$.

Example

For examples of routine usage, see these code examples in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- `cblas_gemm_f16f16f32: examples\cblas\source\cblas_gemm_f16f16f32x.c`

Application Notes

On architectures without native half precision hardware instructions, matrix *A* and *B* are upconverted to single precision and `SGEMM` is called to compute matrix multiplication operation.

cblas_?gemm_free

Frees the storage previously allocated for the packed matrix (deprecated).

Syntax

```
void cblas_sgemm_free (float *dest);  
void cblas_dgemm_free (double *dest);
```

Include Files

- `mkl.h`

Description

The `cblas_?gemm_free` routine is one of a set of related routines that enable use of an internal packed storage. Call the `cblas_?gemm_free` routine last to release storage for the packed matrix structure allocated with `cblas_?gemm_alloc (deprecated)`.

Input Parameters

dest Previously allocated storage.

Output Parameters

dest The freed buffer.

See Also

`cblas_?gemm_pack` Performs scaling and packing of the matrix into the previously allocated buffer.
`cblas_?gemm_compute` Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.

`cblas_?gemm`
 for a detailed description of general matrix multiplication.

`cblas_gemm_*`

Computes a matrix-matrix product with general integer matrices.

Syntax

```
void cblas_gemm_s8u8s32 (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const CBLAS_OFFSET offsetc, const MKL_INT m, const MKL_INT n,
const MKL_INT k, const float alpha, const void *a, const MKL_INT lda, const MKL_INT8
oa, const void *b, const MKL_INT ldb, const MKL_INT8 ob, const float beta, MKL_INT32 *c,
const MKL_INT ldc, const MKL_INT32 *oc);
```

```
void cblas_gemm_s16s16s32 (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa,
const CBLAS_TRANSPOSE transb, const CBLAS_OFFSET offsetc, const MKL_INT m, const
MKL_INT n, const MKL_INT k, const float alpha, const MKL_INT16 *a, const MKL_INT lda,
const MKL_INT16 oa, const MKL_INT16 *b, const MKL_INT ldb, const MKL_INT16 ob, const
float beta, MKL_INT32 *c, const MKL_INT ldc, const MKL_INT32 *oc);
```

Include Files

- `mk1.h`

Description

The `cblas_gemm_*` routines compute a scalar-matrix-matrix product and adds the result to a scalar-matrix product. To get the final result, a vector is added to each row or column of the output matrix. The operation is defined as:

$$C := \alpha * (\text{op}(A) + A_offset) * (\text{op}(B) + B_offset) + \beta * C + C_offset$$

where :

- $\text{op}(X)$ is either $\text{op}(X) = X$ or $\text{op}(X) = X^T$,
- A_offset is an m -by- k matrix with every element equal to the value oa ,
- B_offset is a k -by- n matrix with every element equal to the value ob ,
- C_offset is an m -by- n matrix defined by the oc array as described in the description of the *offsetc* parameter,
- α and β are scalars,
- A is a matrix such that $\text{op}(A)$ is m -by- k ,
- B is a matrix such that $\text{op}(B)$ is k -by- n ,
- and C is an m -by- n matrix.

Input Parameters

Layout Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

transa Specifies the form of $\text{op}(A)$ used in the matrix multiplication:

if *transa*=CblasNoTrans, then $\text{op}(A) = A$;

if *transa*=CblasTrans, then $\text{op}(A) = A^T$.

transb

Specifies the form of $\text{op}(B)$ used in the matrix multiplication:

if *transb*=CblasNoTrans, then $\text{op}(B) = B$;

if *transb*=CblasTrans, then $\text{op}(B) = B^T$.

offsetc

Specifies the form of *C_offset* used in the matrix multiplication.

offsetc = CblasFixOffset: *oc* has a single element and every element of *C_offset* is equal to this element.

offsetc = CblasColOffset: *oc* has a size of *m* and every column of *C_offset* is equal to *oc*.

offsetc = CblasRowOffset: *oc* has a size of *n* and every row of *C_offset* is equal to *oc*.

m

Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix *C*. The value of *m* must be at least zero.

n

Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix *C*. The value of *n* must be at least zero.

k

Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of *k* must be at least zero.

alpha

. Specifies the scalar *alpha*.

a

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans
<i>Layout</i> = CblasColMajor	Array, size <i>lda</i> * <i>k</i> Before entry, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> of 8-bit signed integers for <code>cblas_gemm_s8u8s32</code> or 16-bit signed integers for <code>cblas_gemm_s16s16s32</code> .	Array, size <i>lda</i> * <i>m</i> Before entry, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i> of 8-bit signed integers for <code>cblas_gemm_s8u8s32</code> or 16-bit signed integers for <code>cblas_gemm_s16s16s32</code> .
<i>Layout</i> = CblasRowMajor	Array, size <i>lda</i> * <i>m</i> Before entry, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i> of 8-bit unsigned integers for <code>cblas_gemm_s8u8s32</code> or 16-bit signed integers for <code>cblas_gemm_s16s16s32</code> .	Array, size <i>lda</i> * <i>k</i> Before entry, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> of 8-bit unsigned integers for <code>cblas_gemm_s8u8s32</code> or 16-bit signed integers for <code>cblas_gemm_s16s16s32</code> .

lda

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

	<i>transa</i> =CblasNoTrans	<i>transa</i> =CblasTrans
--	-----------------------------	---------------------------

<i>Layout</i> = CblasColMajor	<i>lda</i> must be at least $\max(1, m)$.	<i>lda</i> must be at least $\max(1, k)$.
<i>Layout</i> = CblasRowMajor	<i>lda</i> must be at least $\max(1, k)$.	<i>lda</i> must be at least $\max(1, m)$.

*oa*Specifies the scalar offset value for matrix *A*.*b*

	<i>transb</i> =CblasNoTrans	<i>transb</i> =CblasTrans
<i>Layout</i> = CblasColMajor	Array, size <i>ldb</i> by <i>n</i> Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> of 8-bit unsigned integers for <code>cblas_gemm_s8u8s32</code> or 16-bit signed integers for <code>cblas_gemm_s16s16s32</code> .	Array, size <i>ldb</i> by <i>k</i> Before entry the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> of 8-bit unsigned integers for <code>cblas_gemm_s8u8s32</code> or 16-bit signed integers for <code>cblas_gemm_s16s16s32</code> .
<i>Layout</i> = CblasRowMajor	Array, size <i>ldb</i> by <i>k</i> Before entry the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> of 8-bit signed integers for <code>cblas_gemm_s8u8s32</code> or 16-bit signed integers for <code>cblas_gemm_s16s16s32</code> .	Array, size <i>ldb</i> by <i>n</i> Before entry, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> of 8-bit signed integers for <code>cblas_gemm_s8u8s32</code> or 16-bit signed integers for <code>cblas_gemm_s16s16s32</code> .

*ldb*Specifies the leading dimension of *b* as declared in the calling (sub)program.

	<i>transb</i> =CblasNoTrans	<i>transb</i> =CblasTrans
<i>Layout</i> = CblasColMajor	<i>ldb</i> must be at least $\max(1, k)$.	<i>ldb</i> must be at least $\max(1, n)$.
<i>Layout</i> = CblasRowMajor	<i>ldb</i> must be at least $\max(1, n)$.	<i>ldb</i> must be at least $\max(1, k)$.

*ob*Specifies the scalar offset value for matrix *B*.*beta*Specifies the scalar *beta*. When *beta* is equal to zero, then *c* need not be set on input.*c*

<i>Layout</i> = CblasColMajor	Array, size <i>ldc</i> by <i>n</i> . Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.
----------------------------------	--

<i>Layout</i> = CblasRowMajor	Array, size <i>ldc</i> by <i>m</i> . Before entry, the leading <i>n</i> -by- <i>m</i> part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.
----------------------------------	--

ldc

Specifies the leading dimension of *c* as declared in the calling (sub)program.

<i>Layout</i> = CblasColMajor	<i>ldc</i> must be at least $\max(1, m)$.
<i>Layout</i> = CblasRowMajor	<i>ldc</i> must be at least $\max(1, n)$.

oc

Array, size *len*. Specifies the offset values for matrix *C*.

If *offsetc* = CblasFixOffset: *len* must be at least 1.

If *offsetc* = CblasColOffset: *len* must be at least $\max(1, m)$.

If *offsetc* = CblasRowOffset: *oc* must be at least $\max(1, n)$.

Output Parameters

c

Overwritten by $\alpha * (\text{op}(A) + A_offset) * (\text{op}(B) + B_offset) + \beta * C + C_offset$.

Example

For examples of routine usage, see the code in the following links and in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- `cblas_gemm_s8u8s32: examples\cblas\source\cblas_gemm_s8u8s32x.c`
- `cblas_gemm_s16s16s32: examples\cblas\source\cblas_gemm_s16s16s32x.c`

Application Notes

The matrix-matrix product can be expanded:

$$(\text{op}(A) + A_offset) * (\text{op}(B) + B_offset)$$

$$= \text{op}(A) * \text{op}(B) + \text{op}(A) * B_offset + A_offset * \text{op}(B) + A_offset * B_offset$$

After computing these four multiplication terms separately, they are summed from left to right. The results from the matrix-matrix product and the *C* matrix are scaled with *alpha* and *beta* floating-point values respectively using double-precision arithmetic. Before storing the results to the output *c* array, the floating-point values are rounded to the nearest integers. In the event of overflow or underflow, the results depend on the architecture. The results are either unsaturated (wrapped) or saturated to maximum or minimum representable integer values for the data type of the output matrix.

When using `cblas_gemm_s8u8s32` with row-major layout, the data types of *A* and *B* must be swapped. That is, you must provide an 8-bit unsigned integer array for matrix *A* and an 8-bit signed integer array for matrix *B*.

Intermediate integer computations in `cblas_gemm_s8u8s32` on 64-bit Intel® Advanced Vector Extensions 2 (Intel® AVX2) and Intel® Advanced Vector Extensions 512 (Intel® AVX-512) architectures without Vector Neural Network Instructions (VNNI) extensions can saturate. This is because only 16-bits are available for the accumulation of intermediate results. You can avoid integer saturation by maintaining all integer elements of *A* or *B* matrices under 8 bits.

cblas_?gemv_batch_strided

Computes groups of matrix-vector product with general matrices.

Syntax

```
void cblas_sgemv_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE trans,
const MKL_INT m, const MKL_INT n, const float alpha, const float *a, const MKL_INT lda,
const MKL_INT stridea, const float *x, const MKL_INT incx, const MKL_INT stridex, const
float beta, float *y, const MKL_INT incy, const MKL_INT stridey, const MKL_INT
batch_size);
```

```
void cblas_dgemv_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE trans,
const MKL_INT m, const MKL_INT n, const double alpha, const double *a, const MKL_INT
lda, const MKL_INT stridea, const double *x, const MKL_INT incx, const MKL_INT stridex,
const double beta, double *y, const MKL_INT incy, const MKL_INT stridey, const MKL_INT
batch_size);
```

```
void cblas_cgemv_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE trans,
const MKL_INT m, const MKL_INT n, const void alpha, const void *a, const MKL_INT lda,
const MKL_INT stridea, const void *x, const MKL_INT incx, const MKL_INT stridex, const
void beta, void *y, const MKL_INT incy, const MKL_INT stridey, const MKL_INT
batch_size);
```

```
void cblas_zgemv_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE trans,
const MKL_INT m, const MKL_INT n, const void alpha, const void *a, const MKL_INT lda,
const MKL_INT stridea, const void *x, const MKL_INT incx, const MKL_INT stridex, const
void beta, void *y, const MKL_INT incy, const MKL_INT stridey, const MKL_INT
batch_size);
```

Include Files

- mkl.h

Description

The `cblas_?gemv_batch_strided` routines perform a series of matrix-vector product added to a scaled vector. They are similar to the `cblas_?gemv` routine counterparts, but the `cblas_?gemv_batch_strided` routines perform matrix-vector operations with groups of matrices and vectors.

All matrices *a* and vectors *x* and *y* have the same parameters (size, increments) and are stored at constant *stridea*, *stridex*, and *stridey* from each other. The operation is defined as

```
for i = 0 ... batch_size - 1
    A is a matrix at offset i * stridea in a
    X and Y are vectors at offset i * stridex and i * stridey in x and y
    Y = alpha * op(A) * X + beta * Y
end for
```

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>trans</i>	Specifies <code>op(A)</code> the transposition operation applied to the <i>A</i> matrices. if <i>trans</i> = CblasNoTrans, then <code>op(A) = A</code> ; if <i>trans</i> = CblasTrans, then <code>op(A) = A'</code> ;

	if <i>trans</i> = CblasConjTrans, then $\text{op}(A) = \text{conj}(A')$.
<i>m</i>	Number of rows of the matrices <i>A</i> . The value of <i>m</i> must be at least 0.
<i>n</i>	Number of columns of the matrices <i>A</i> . The value of <i>n</i> must be at least 0.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .
<i>a</i>	Array holding all the input matrix <i>A</i> . Must be of size at least $\text{lda} * k + \text{stridea} * (\text{batch_size} - 1)$ where <i>k</i> is <i>n</i> if column major layout is used or <i>m</i> if row major layout is used.
<i>lda</i>	Specifies the leading dimension of the matrix <i>A</i> . It must be positive and at least <i>m</i> if column major layout is used or at least <i>n</i> if row major layout is used.
<i>stridea</i>	Stride between two consecutive <i>A</i> matrices. Must be at least 0.
<i>x</i>	Array holding all the input vector <i>x</i> . Must be of size at least $(1 + (\text{len} - 1) * \text{abs}(\text{incx})) + \text{stridex} * (\text{batch_size} - 1)$ where <i>len</i> is <i>n</i> if the <i>A</i> matrix is not transposed or <i>m</i> otherwise.
<i>incx</i>	Stride between two consecutive elements of the <i>x</i> vectors. Must not be zero.
<i>stridex</i>	Stride between two consecutive <i>x</i> vectors, must be at least 0.
<i>beta</i>	Specifies the scalar <i>beta</i> .
<i>y</i>	Array holding all the input vectors <i>y</i> . Must be of size at least $\text{batch_size} * \text{stridey}$.
<i>incy</i>	Stride between two consecutive elements of the <i>y</i> vectors. Must not be zero.
<i>stridey</i>	Stride between two consecutive <i>y</i> vectors, must be at least $(1 + (\text{len} - 1) * \text{abs}(\text{incy}))$ where <i>len</i> is <i>m</i> if the matrix <i>A</i> is non transpose or <i>n</i> otherwise.
<i>batch_size</i>	Number of <code>gemv</code> computations to perform and <i>a</i> matrices, <i>x</i> and <i>y</i> vectors. Must be at least 0.

Output Parameters

<i>y</i>	Array holding the <i>batch_size</i> updated vector <i>y</i> .
----------	---

cblas_?gemv_batch

Computes groups of matrix-vector product with general matrices.

Syntax

```
void cblas_sgemv_batch (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE *trans_array,
const MKL_INT *m_array, const MKL_INT *n_array, const float *alpha_array, const float
**a_array, const MKL_INT *lda_array, const float **x_array, const MKL_INT *incx_array,
const float *beta_array, float **y_array, const MKL_INT *incy_array, const MKL_INT
group_count, const MKL_INT *group_size);
```

```

void cblas_dgemv_batch (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE *trans_array,
const MKL_INT *m_array, const MKL_INT *n_array, const double *alpha_array, const double
**a_array, const MKL_INT *lda_array, const double **x_array, const MKL_INT *incx_array,
const double *beta_array, double **y_array, const MKL_INT *incy_array, const MKL_INT
group_count, const MKL_INT *group_size);

void cblas_cgemv_batch (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE *trans_array,
const MKL_INT *m_array, const MKL_INT *n_array, const void *alpha_array, const void
**a_array, const MKL_INT *lda_array, const void **x_array, const MKL_INT *incx_array,
const void *beta_array, void **y_array, const MKL_INT *incy_array, const MKL_INT
group_count, const MKL_INT *group_size);

void cblas_zgemv_batch (const CBLAS_LAYOUT layout, const CBLAS_TRANSPOSE *trans_array,
const MKL_INT *m_array, const MKL_INT *n_array, const void *alpha_array, const void
**a_array, const MKL_INT *lda_array, const void **x_array, const MKL_INT *incx_array,
const void *beta_array, void **y_array, const MKL_INT *incy_array, const MKL_INT
group_count, const MKL_INT *group_size);

```

Include Files

- mkl.h

Description

The `cblas_?gemv_batch` routines perform a series of matrix-vector product added to a scaled vector. They are similar to the `cblas_?gemv` routine counterparts, but the `cblas_?gemv_batch` routines perform matrix-vector operations with groups of matrices and vectors.

Each group contains matrices and vectors with the same parameters (size, increments). The operation is defined as:

```

idx = 0
For i = 0 ... group_count - 1
    trans, m, n, alpha, lda, incx, beta, incy and group_size at position i in trans_array,
m_array, n_array, alpha_array, lda_array, incx_array, beta_array, incy_array and group_size_array
    for j = 0 ... group_size - 1
        a is a matrix of size mxn at position idx in a_array
        x and y are vectors of size m or n depending on trans, at position idx in x_array and
y_array
        y := alpha * op(a) * x + beta * y
        idx := idx + 1
    end for
end for

```

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

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>trans_array</i>	<p>Array of size <code>group_count</code>. For the group <i>i</i>, <code>trans_i = trans_array[i]</code> specifies the transposition operation applied to A.</p> <p>if <code>trans = CblasNoTrans</code>, then <code>op(A) = A</code>;</p> <p>if <code>trans = CblasTrans</code>, then <code>op(A) = A'</code>;</p> <p>if <code>trans = CblasConjTrans</code>, then <code>op(A) = conjg(A')</code>.</p>

<i>m_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , $m_i = m_array[i]$ is the number of rows of the matrix <i>A</i> .
<i>n_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , $n_i = n_array[i]$ is the number of columns in the matrix <i>A</i> .
<i>alpha_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , $alpha_i = alpha_array[i]$ is the scalar <i>alpha</i> .
<i>a_array</i>	Array of size <i>total_batch_count</i> of pointers used to store <i>A</i> matrices. The array allocated for the <i>A</i> matrices of the group <i>i</i> must be of size at least $lda_i * n_i$ if column major layout is used or at least $lda_i * m_i$ if row major layout is used.
<i>lda_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , $lda_i = lda_array[i]$ is the leading dimension of the matrix <i>A</i> . It must be positive and at least m_i if column major layout is used or at least n_i if row major layout is used..
<i>x_array</i>	Array of size <i>total_batch_count</i> of pointers used to store <i>x</i> vectors. The array allocated for the <i>x</i> vectors of the group <i>i</i> must be of size at least $(1 + len_i - 1) * abs(incx_i)$ where len_i is n_i if the <i>A</i> matrix is not transposed or m_i otherwise.
<i>incx_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , $incx_i = incx_array[i]$ is the stride of vector <i>x</i> . Must not be zero.
<i>beta_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , $beta_i = beta_array[i]$ is the scalar <i>beta</i> .
<i>y_array</i>	Array of size <i>total_batch_count</i> of pointers used to store <i>y</i> vectors. The array allocated for the <i>y</i> vectors of the group <i>i</i> must be of size at least $(1 + len_i - 1) * abs(incy_i)$ where len_i is m_i if the <i>A</i> matrix is not transposed or n_i otherwise.
<i>incy_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , $incy_i = incy_array[i]$ is the stride of vector <i>y</i> . Must not be zero.
<i>group_count</i>	Number of groups. Must be at least 0.
<i>group_size</i>	Array of size <i>group_count</i> . The element <i>group_count</i> [<i>i</i>] is the number of operations in the group <i>i</i> . Each element in <i>group_count</i> must be at least 0.

Output Parameters

<i>y_array</i>	Array of pointers holding the <i>total_batch_count</i> updated vector <i>y</i> .
----------------	--

cblas_dgmm_batch_strided

Computes groups of matrix-vector product using general matrices.

Syntax

```
void cblas_sdgmm_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_SIDE left_right,
const MKL_INT m, const MKL_INT n, const float *a, const MKL_INT lda, const MKL_INT
stridea, const float *x, const MKL_INT incx, const MKL_INT stridex, const float *c,
const MKL_INT ldc, const MKL_INT stridec, const MKL_INT batch_size);
```

```

void cblas_ddgmm_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_SIDE left_right,
const MKL_INT m, const MKL_INT n, const double *a, const MKL_INT lda, const MKL_INT
stridea, const double *x, const MKL_INT incx, const MKL_INT stridex, const double *c,
const MKL_INT ldc, const MKL_INT stridec, const MKL_INT batch_size);

void cblas_cdgmm_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_SIDE left_right,
const MKL_INT m, const MKL_INT n, const void *a, const MKL_INT lda, const MKL_INT
stridea, const void *x, const MKL_INT incx, const MKL_INT stridex, const void *c, const
MKL_INT ldc, const MKL_INT stridec, const MKL_INT batch_size);

void cblas_zdgmm_batch_strided (const CBLAS_LAYOUT layout, const CBLAS_SIDE left_right,
const MKL_INT m, const MKL_INT n, const void *a, const MKL_INT lda, const MKL_INT
stridea, const void *x, const MKL_INT incx, const MKL_INT stridex, const void *c, const
MKL_INT ldc, const MKL_INT stridec, const MKL_INT batch_size);

```

Include Files

- mkl.h

Description

The `cblas_?dgmm_batch_strided` routines perform a series of diagonal matrix-matrix product. The diagonal matrices are stored as dense vectors and the operations are performed with group of matrices and vectors.

All matrices *a* and *c* and vector *x* have the same parameters (size, increments) and are stored at constant stride, respectively, given by *stridea*, *stridec*, and *stridex* from each other. The operation is defined as

```

for i = 0 ... batch_size - 1
    A and C are matrices at offset i * stridea in a and i * stridec in c
    X is a vector at offset i * stridex in x
    C = diag(X) * A or C = A * diag(X)
end for

```

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>left_right</i>	Specifies the position of the diagonal matrix in the matrix product if <i>left_right</i> = CblasLeft, then $C = \text{diag}(X) * A$; if <i>left_right</i> = CblasRight, then $C = A * \text{diag}(X)$.
<i>m</i>	Number of rows of the matrices <i>A</i> and <i>C</i> . The value of <i>m</i> must be at least 0.
<i>n</i>	Number of columns of the matrices <i>A</i> and <i>C</i> . The value of <i>n</i> must be at least 0.
<i>a</i>	Array holding all the input matrix <i>A</i> . Must be of size at least $lda * k + stridea * (batch_size - 1)$ where <i>k</i> is <i>n</i> if column major layout is used or <i>m</i> if row major layout is used.
<i>lda</i>	Specifies the leading dimension of the matrix <i>A</i> . It must be positive and at least <i>m</i> if column major layout is used or at least <i>n</i> if row major layout is used.
<i>stridea</i>	Stride between two consecutive <i>A</i> matrices, must be at least 0.

<code>x</code>	Array holding all the input vector <code>x</code> . Must be of size at least $(1 + (len - 1) * abs(incx)) + stridex * (batch_size - 1)$ where <code>len</code> is <code>n</code> if the diagonal matrix is on the right of the product or <code>m</code> otherwise.
<code>incx</code>	Stride between two consecutive elements of the <code>x</code> vectors.
<code>stridex</code>	Stride between two consecutive <code>x</code> vectors, must be at least 0.
<code>c</code>	Array holding all the input matrix <code>C</code> . Must be of size at least <code>batch_size * stridec</code> .
<code>ldc</code>	Specifies the leading dimension of the matrix <code>C</code> . It must be positive and at least <code>mif</code> column major layout is used or at least <code>n</code> if row major layout is used.
<code>stridec</code>	Stride between two consecutive <code>A</code> matrices, must be at least <code>ldc * nif</code> column major layout is used or <code>ldc * m</code> if row major layout is used.
<code>batch_size</code>	Number of <code>dgmm</code> computations to perform and a <code>c</code> matrices and <code>x</code> vectors. Must be at least 0.

Output Parameters

<code>c</code>	Array holding the <code>batch_size</code> updated matrices <code>c</code> .
----------------	---

cblas_?dgmm_batch

Computes groups of matrix-vector product using general matrices.

Syntax

```
void cblas_sdgmm_batch (const CBLAS_LAYOUT layout, const CBLAS_SIDE *left_right_array,
const MKL_INT *m_array, const MKL_INT *n_array, const float **a_array, const MKL_INT
*lda_array, const float **x_array, const MKL_INT *incx_array, float **c_array, const
MKL_INT *ldc_array, const MKL_INT group_count, const MKL_INT *group_size);

void cblas_ddgmm_batch (const CBLAS_LAYOUT layout, const CBLAS_SIDE *left_right_array,
const MKL_INT *m_array, const MKL_INT *n_array, const double **a_array, const MKL_INT
*lda_array, const double **x_array, const MKL_INT *incx_array, double **c_array, const
MKL_INT *ldc_array, const MKL_INT group_count, const MKL_INT *group_size);

void cblas_cdgmm_batch (const CBLAS_LAYOUT layout, const CBLAS_SIDE *left_right_array,
const MKL_INT *m_array, const MKL_INT *n_array, const void **a_array, const MKL_INT
*lda_array, const void **x_array, const MKL_INT *incx_array, void **c_array, const
MKL_INT *ldc_array, const MKL_INT group_count, const MKL_INT *group_size);

void cblas_zdgmm_batch (const CBLAS_LAYOUT layout, const CBLAS_SIDE *left_right_array,
const MKL_INT *m_array, const MKL_INT *n_array, const void **a_array, const MKL_INT
*lda_array, const void **x_array, const MKL_INT *incx_array, void **c_array, const
MKL_INT *ldc_array, const MKL_INT group_count, const MKL_INT *group_size);
```

Include Files

- `mkl.h`

Description

The `cblas_?dgmm_batch` routines perform a series of diagonal matrix-matrix product. The diagonal matrices are stored as dense vectors and the operations are performed with group of matrices and vectors. .

Each group contains matrices and vectors with the same parameters (size, increments). The operation is defined as:

```

idx = 0
For i = 0 ... group_count - 1
    left_right, m, n, lda, incx, ldc and group_size at position i in left_right_array, m_array,
    n_array, lda_array, incx_array, ldc_array and group_size_array
    for j = 0 ... group_size - 1
        a and c are matrices of size mxn at position idx in a_array and c_array
        x is a vector of size m or n depending on left_right, at position idx in x_array
        if (left_right == oneapi::mkl::side::left) c := diag(x) * a
        else c := a * diag(x)
        idx := idx + 1
    end for
end for

```

The number of entries in *a_array*, *x_array*, and *c_array* is *total_batch_count* = the sum of all of the *group_size* entries.

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>left_right_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , <i>left_right_i</i> = <i>left_right_array</i> [<i>i</i>] specifies the position of the diagonal matrix in the matrix product. if <i>left_right_i</i> = CblasLeft, then $C = \text{diag}(X) * A$. if <i>left_right_i</i> = CblasRight, then $C = A * \text{diag}(X)$.
<i>m_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , <i>m_i</i> = <i>m_array</i> [<i>i</i>] is the number of rows of the matrix <i>A</i> and <i>C</i> .
<i>n_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , <i>n_i</i> = <i>n_array</i> [<i>i</i>] is the number of columns in the matrix <i>A</i> and <i>C</i> .
<i>a_array</i>	Array of size <i>total_batch_count</i> of pointers used to store <i>A</i> matrices. The array allocated for the <i>A</i> matrices of the group <i>i</i> must be of size at least <i>lda_i</i> * <i>n_i</i> if column major layout is used or at least <i>lda_i</i> * <i>m_i</i> if row major layout is used.
<i>lda_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , <i>lda_i</i> = <i>lda_array</i> [<i>i</i>] is the leading dimension of the matrix <i>A</i> . It must be positive and at least <i>m_i</i> if column major layout is used or at least <i>n_i</i> if row major layout is used..
<i>x_array</i>	Array of size <i>total_batch_count</i> of pointers used to store <i>x</i> vectors. The array allocated for the <i>x</i> vectors of the group <i>i</i> must be of size at least $(1 + \text{len}_i - 1) * \text{abs}(\text{incx}_i)$ where <i>len_i</i> is <i>n_i</i> if the diagonal matrix is on the right of the product or <i>m_i</i> otherwise.
<i>incx_array</i>	Array of size <i>group_count</i> . For the group <i>i</i> , <i>incx_i</i> = <i>incx_array</i> [<i>i</i>] is the stride of vector <i>x</i> .
<i>c_array</i>	Array of size <i>total_batch_count</i> of pointers used to store <i>C</i> matrices. The array allocated for the <i>C</i> matrices of the group <i>i</i> must be of size at least <i>ldc_i</i> * <i>n_i</i> , if column major layout is used or at least <i>ldc_i</i> * <i>m_i</i> if row major layout is used.

<code>ldc_array</code>	Array of size <code>group_count</code> . For the group i , $ldc_i = ldc_array[i]$ is the leading dimension of the matrix C . It must be positive and at least m_i if column major layout is used or at least n_i if row major layout is used..
<code>group_count</code>	Number of groups. Must be at least 0.
<code>group_size</code>	Array of size <code>group_count</code> . The element <code>group_count[i]</code> is the number of operations in the group i . Each element in <code>group_size</code> must be at least 0.

Output Parameters

<code>c_array</code>	Array of pointers holding the <code>total_batch_count</code> updated matrix C .
----------------------	---

mkl_jit_create_?gemm

Create a GEMM kernel that computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product.

Syntax

```
mkl_jit_status_t mkl_jit_create_sgemm(void** jitter, const MKL_LAYOUT layout, const
MKL_TRANPOSE transa, const MKL_TRANSPOSE transb, const MKL_INT m, const MKL_INT n,
const MKL_INT k, const float alpha, const MKL_INT lda, const MKL_INT ldb, const float
beta, const MKL_INT ldc);

mkl_jit_status_t mkl_jit_create_dgemm(void** jitter, const MKL_LAYOUT layout, const
MKL_TRANPOSE transa, const MKL_TRANSPOSE transb, const MKL_INT m, const MKL_INT n,
const MKL_INT k, const double alpha, const MKL_INT lda, const MKL_INT ldb, const double
beta, const MKL_INT ldc);

mkl_jit_status_t mkl_jit_create_cgemm(void** jitter, const MKL_LAYOUT layout, const
MKL_TRANPOSE transa, const MKL_TRANSPOSE transb, const MKL_INT m, const MKL_INT n,
const MKL_INT k, const void* alpha, const MKL_INT lda, const MKL_INT ldb, const void*
beta, const MKL_INT ldc);

mkl_jit_status_t mkl_jit_create_zgemm(void** jitter, const MKL_LAYOUT layout, const
MKL_TRANPOSE transa, const MKL_TRANSPOSE transb, const MKL_INT m, const MKL_INT n,
const MKL_INT k, const void* alpha, const MKL_INT lda, const MKL_INT ldb, const void*
beta, const MKL_INT ldc);
```

Include Files

- `mkl.h`

Description

The `mkl_jit_create_?gemm` functions belong to a set of related routines that enable use of just-in-time code generation.

The `mkl_jit_create_?gemm` functions create a handle to a just-in-time code generator (a jitter) and generate a GEMM kernel that computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product, with general matrices. The operation of the generated GEMM kernel is defined as follows:

```
C := alpha*op(A)*op(B) + beta*C
```

Where:

- `op(X)` is either `op(X) = X` or `op(X) = XT` or `op(X) = XH`
- `alpha` and `beta` are scalars

- A , B , and C are matrices
- $\text{op}(A)$ is an m -by- k matrix
- $\text{op}(B)$ is a k -by- n matrix
- C is an m -by- n matrix

NOTE

Generating a new kernel with `mkl_jit_create_?gemm` involves moderate runtime overhead. To benefit from JIT code generation, use this feature when you need to call the generated kernel many times (for example, several hundred calls).

Input Parameters

<i>layout</i>	Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).
<i>transa</i>	Specifies the form of $\text{op}(A)$ used in the generated matrix multiplication: <ul style="list-style-type: none"> • if <i>transa</i> = MKL_NOTRANS, then $\text{op}(A) = A$ • if <i>transa</i> = MKL_TRANS, then $\text{op}(A) = A^T$ • if <i>transa</i> = MKL_CONJTRANS, then $\text{op}(A) = A^H$
<i>transb</i>	Specifies the form of $\text{op}(B)$ used in the generated matrix multiplication: <ul style="list-style-type: none"> • if <i>transb</i> = MKL_NOTRANS, then $\text{op}(B) = B$ • if <i>transb</i> = MKL_TRANS, then $\text{op}(B) = B^T$ • if <i>transb</i> = MKL_CONJTRANS, then $\text{op}(B) = B^H$
<i>m</i>	Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C . The value of <i>m</i> must be at least zero.
<i>n</i>	Specifies the number of columns of the matrix $\text{op}(B)$ and of the matrix C . The value of <i>n</i> must be at least zero.
<i>k</i>	Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of <i>k</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> .

NOTE

alpha is passed by pointer for `mkl_jit_create_cgemm` and `mkl_jit_create_zgemm`.

lda Specifies the leading dimension of *a*.

	<i>transa</i> =MKL_NOTRANS	<i>transa</i> =MKL_TRANS or <i>transa</i> =MKL_CONJTRANS
<i>layout</i> =MKL_ROW_MAJOR	<i>lda</i> must be at least $\max(1,k)$	<i>lda</i> must be at least $\max(1,m)$
<i>layout</i> =MKL_COL_MAJOR	<i>lda</i> must be at least $\max(1,m)$	<i>lda</i> must be at least $\max(1,k)$

*ldb*Specifies the leading dimension of *b*:

	<i>transb</i> =MKL_NOTRAN S	<i>transb</i> =MKL_TRANS or <i>transb</i> =MKL_CONJTRANS
<i>layout</i> =MKL_ROW_MAJOR	<i>ldb</i> must be at least max(1,n)	<i>ldb</i> must be at least max(1,k)
<i>layout</i> =MKL_COL_MAJOR	<i>ldb</i> must be at least max(1,k)	<i>ldb</i> must be at least max(1,n)

*beta*Specifies the scalar *beta*.**NOTE**

beta is passed by pointer for `mkl_jit_create_cgemm` and `mkl_jit_create_zgemm`.

*ldc*Specifies the leading dimension of *c*.

<i>layout</i> =MKL_ROW_MAJOR	<i>ldc</i> must be at least max(1,n)
<i>layout</i> =MKL_COL_MAJOR	<i>ldc</i> must be at least max(1,m)

Output Parameters*jitter*

Pointer to a handle to the newly created code generator.

Return Values*status*

Returns one of the following:

- MKL_JIT_ERROR if the handle cannot be created (no memory)
—or—
- MKL_JIT_SUCCESS if the jitter has been created and the GEMM kernel was successfully created
—or—
- MKL_NO_JIT if the jitter has been created, but a JIT GEMM kernel was not created because JIT is not beneficial for the given input parameters. The function pointer returned by `mkl_jit_get_?gemm_ptr` will call standard (non-JIT) GEMM.

mkl_jit_get_?gemm_ptr

Return the GEMM kernel associated with a jitter previously created with `mkl_jit_create_?gemm`.

Syntax

```
sgemm_jit_kernel_t mkl_jit_get_sgemm_ptr(const void* jitter);
dgemm_jit_kernel_t mkl_jit_get_dgemm_ptr(const void* jitter);
cgemm_jit_kernel_t mkl_jit_get_cgemm_ptr(const void* jitter);
```

```
zgemm_jit_kernel_t mkl_jit_get_zgemm_ptr(const void* jitter);
```

Include Files

- `mkl.h`

Description

The `mkl_jit_get_?gemm_ptr` functions belong to a set of related routines that enable use of just-in-time code generation.

The `mkl_jit_get_?gemm_ptr` functions take as input a jitter previously created with `mkl_jit_create_?gemm`, and return the GEMM kernel associated with that jitter. The returned GEMM kernel computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product, with general matrices. The operation is defined as follows:

```
C := alpha*op(A)*op(B) + beta*C
```

Where:

- `op(X)` is one of `op(X) = X` or `op(X) = XT` or `op(X) = XH`
- `alpha` and `beta` are scalars
- `A`, `B`, and `C` are matrices
- `op(A)` is an `m-by-k` matrix
- `op(B)` is a `k-by-n` matrix
- `C` is an `m-by-n` matrix

NOTE

Generating a new kernel with `mkl_jit_create_?gemm` involves moderate runtime overhead. To benefit from JIT code generation, use this feature when you need to call the generated kernel many times (for example, several hundred calls).

Input Parameter

`jitter`

Handle to the code generator.

Return Values

`func`

- `sgemm_jit_kernel_t` – A function pointer type expecting four inputs of type `void*`, `float*`, `float*`, and `float*`

```
typedef void (*sgemm_jit_kernel_t)
(void*, float*, float*, float*);
```

- `dgemm_jit_kernel_t` – A function pointer type expecting four inputs of type `void*`, `double*`, `double*`, and `double*`

```
typedef void (*dgemm_jit_kernel_t)
(void*, double*, double*, double*);
```

- `cgemm_jit_kernel_t` – A function pointer type expecting four inputs of type `void*`, `MKL_Complex8*`, `MKL_Complex8*`, and `MKL_Complex8*`

```
typedef void (*cgemm_jit_kernel_t)
(void*, MKL_Complex8*, MKL_Complex8*, MKL_Complex8*);
```

- `zgemm_jit_kernel_t` – A function pointer type expecting four inputs of type `void*`, `MKL_Complex16*`, `MKL_Complex16*`, and `MKL_Complex16*`

```
typedef void(*zgemm_jit_kernel_t)
(void*, MKL_Complex16*, MKL_Complex16*, MKL_Complex16*);
```

If the jitter input is not NULL, returns a function pointer to a GEMM kernel. The GEMM kernel is called with four parameters: the jitter and the three matrices *a*, *b*, and *c*. Otherwise, returns NULL.

If *layout*, *transa*, *transb*, *m*, *n*, *k*, *lda*, *ldb*, and *ldc* are the parameters used during the creation of the input jitter, then:

a

	<i>layout</i> = MKL_COL_MAJOR	<i>layout</i> = MKL_ROW_MAJOR
<i>transa</i> = MKL_NOTRANS	Array of size <i>lda</i> * <i>k</i> Before calling the returned function pointer, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix A.	Array of size <i>lda</i> * <i>m</i> Before calling the returned function pointer, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix A.
<i>transa</i> = MKL_TRANS or <i>transa</i> = MKL_CONJTRANS	Array of size <i>lda</i> * <i>m</i> Before calling the returned function pointer, the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix A.	Array of size <i>lda</i> * <i>k</i> Before calling the returned function pointer, the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix A.

b

	<i>layout</i> = MKL_COL_MAJOR	<i>layout</i> = MKL_ROW_MAJOR
<i>transb</i> = MKL_NOTRANS	Array of size <i>ldb</i> * <i>n</i> Before calling the returned function pointer, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix B.	Array of size <i>ldb</i> * <i>k</i> Before calling the returned function pointer, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix B.
<i>transb</i> = MKL_TRANS or <i>transb</i> = MKL_CONJTRANS	Array of size <i>ldb</i> * <i>k</i> Before calling the returned function pointer, the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix B.	Array of size <i>ldb</i> * <i>n</i> Before calling the returned function pointer, the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix B.

C

<i>layout</i> = MKL_COL_MAJOR	<i>layout</i> = MKL_ROW_MAJOR
Array of size $ldc*n$ Before calling the returned function pointer, the leading m -by- n part of the array <i>c</i> must contain the matrix C.	Array of size $ldc*m$ Before calling the returned function pointer, the leading n -by- m part of the array <i>c</i> must contain the matrix C.

mkl_jit_destroy

Delete the jitter previously created with `mkl_jit_create_?gemm` as well as the GEMM kernel that it contains.

Syntax

```
mkl_jit_status_t mkl_jit_destroy (void* jitter);
```

Include Files

- `mkl.h`

Description

The `mkl_jit_destroy` function belongs to a set of related routines that enable use of just-in-time code generation.

The `mkl_jit_destroy` function takes as input a jitter previously created with `mkl_jit_create_?gemm` and deletes the jitter as well as the GEMM kernel that it contains.

NOTE

Generating a new kernel with `mkl_jit_create_?gemm` involves moderate runtime overhead. To benefit from JIT code generation, use this feature when you need to call the generated kernel many times (for example, several hundred calls).

Input Parameter

`jitter` Jitter handle

Return Values

`status`

Returns one of the following:

- `MKL_JIT_ERROR` if the pointer is not NULL and is not a handle on a jitter—that is, if it was not created with `mkl_jit_create_?gemm`
- or—
- `MKL_JIT_SUCCESS` if the jitter has been successfully destroyed

LAPACK Routines

Intel® oneAPI Math Kernel Library (oneMKL) implements routines from the LAPACK package that are used for solving systems of linear equations, linear least squares problems, eigenvalue and singular value problems, and performing a number of related computational tasks. The library includes LAPACK routines for both real and complex data. Routines are supported for systems of equations with the following types of matrices:

- General
- Banded
- Symmetric or Hermitian positive-definite (full, packed, and rectangular full packed (RFP) storage)
- Symmetric or Hermitian positive-definite banded
- Symmetric or Hermitian indefinite (both full and packed storage)
- Symmetric or Hermitian indefinite banded
- Triangular (full, packed, and RFP storage)
- Triangular banded
- Tridiagonal
- Diagonally dominant tridiagonal.

NOTE

Different arrays used as parameters to Intel® MKL LAPACK routines must not overlap.

Warning

LAPACK routines assume that input matrices do not contain IEEE 754 special values such as `INF` or `NaN` values. Using these special values may cause LAPACK to return unexpected results or become unstable.

Choosing a LAPACK Routine

Intel® oneAPI Math Kernel Library (oneMKL) offers many LAPACK routines, and many perform similar operations. The Intel® oneAPI Math Kernel Library LAPACK Function Finding Advisor helps you understand the differences between routines so that you can choose the appropriate routine for your task:

Intel® oneAPI Math Kernel Library LAPACK Function Finding Advisor: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-function-finding-advisor.html>.

C Interface Conventions for LAPACK Routines

The C interfaces are implemented for most of the Intel® oneAPI Math Kernel Library (oneMKL) LAPACK driver and computational routines.

NaN Checking in LAPACKE

NaN checking can affect the performance of an application. By default, it is ON.

See the [Support Functions](#) section for details on the methods and options to turn NaN check off or back on with LAPACKE:.

Function Prototypes

Intel® oneAPI Math Kernel Library (oneMKL) supports four distinct floating-point precisions. Each corresponding prototype looks similar, usually differing only in the data type. C interface LAPACK function names follow the form `<?><name>[_64]`, where `<?>` is:

- LAPACKE_s for float
- LAPACKE_d for double
- LAPACKE_c for lapack_complex_float
- LAPACKE_z for lapack_complex_double

On 64-bit platforms, Intel® oneAPI Math Kernel Library (oneMKL) provides LAPACK C interfaces with the _64 suffix to support large data arrays in the LP64 interface library. For more interface library details, see "Using the ILP64 Interface vs. LP64 Interface" in the developer guide.

A specific example follows. To solve a system of linear equations with a packed Cholesky-factored Hermitian positive-definite matrix with complex precision, use the following:

```
lapack_int LAPACKE_cppttrs(int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* ap, lapack_complex_float* b, lapack_int ldb);
```

For matrices whose dimensions are greater than $2^{31}-1$, you can use either LAPACKE_cppttrs in the ILP64 interface library or LAPACKE_cppttrs_64 in the LP64 interface library.

Workspace Arrays

In contrast to the Fortran interface, the LAPACK C interface omits workspace parameters because workspace is allocated during runtime and released upon completion of the function operation.

If you prefer to allocate workspace arrays yourself, the LAPACK C interface provides alternate interfaces with *work* parameters. The name of the alternate interface is the same as the LAPACK C interface with *_work* appended. For example, the syntax for the singular value decomposition of a real bidiagonal matrix is:

Fortran:	<code>call sbdsdc (uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)</code>
C LAPACK interface:	<code>lapack_int LAPACKE_sbdsdc (int matrix_layout, char uplo, char compq, lapack_int n, float* d, float* e, float* u, lapack_int ldu, float* vt, lapack_int ldvt, float* q, lapack_int* iq);</code>
Alternate C LAPACK interface with <i>work</i> parameters:	<code>lapack_int LAPACKE_sbdsdc_work(int matrix_layout, char uplo, char compq, lapack_int n, float* d, float* e, float* u, lapack_int ldu, float* vt, lapack_int ldvt, float* q, lapack_int* iq, float* work, lapack_int* iwork);</code>

See the `install_dir/include/mkl_lapacke.h` file for the full list of alternative C LAPACK interfaces.

The Intel® oneAPI Math Kernel Library (oneMKL) Fortran-specific documentation contains details about workspace arrays.

Mapping Fortran Data Types against C Data Types

Fortran Data Types vs. C Data Types

FORTRAN	C
INTEGER	lapack_int
LOGICAL	lapack_logical
REAL	float
DOUBLE PRECISION	double
COMPLEX	lapack_complex_float
COMPLEX*16/DOUBLE COMPLEX	lapack_complex_double

FORTTRAN	C
CHARACTER	char

C Type Definitions

You can find type definitions specific to Intel® oneAPI Math Kernel Library (oneMKL) such as `MKL_INT`, `MKL_Complex8`, and `MKL_Complex16` in `install_dir/mkl_types.h`.

C types

```
#ifndef lapack_int
#define lapack_int MKL_INT
#endif

#ifndef lapack_logical
#define lapack_logical lapack_int
#endif
```

Complex Type Definitions

Complex type for single precision:

```
#ifndef lapack_complex_float
#define lapack_complex_float MKL_Complex8
#endif
```

Complex type for double precision:

```
#ifndef lapack_complex_double
#define lapack_complex_double MKL_Complex16
#endif
```

Matrix Layout Definitions

```
#define LAPACK_ROW_MAJOR 101
#define LAPACK_COL_MAJOR 102
```

See [Matrix Layout for LAPACK Routines](#) above for an explanation of row-major order and column-major order storage.

Error Code Definitions

```
#define LAPACK_WORK_MEMORY_ERROR -1010 /* Failed to allocate
memory
                                     for a working array */
#define LAPACK_TRANSPOSE_MEMORY_ERROR -1011 /* Failed to allocate
memory
                                     for transposed matrix */
```

Matrix Layout for LAPACK Routines

There are two general methods of storing a two dimensional matrix in linear (one dimensional) memory: column-wise (column major order) or row-wise (row major order). Consider an M -by- N matrix A :

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,j_0+1} & a_{1,j_0+2} & \dots & a_{1,j_0+L-1} & a_{1,j_0+L} & \dots & a_{1,N} \\ a_{2,1} & a_{2,2} & \dots & a_{2,j_0+1} & a_{2,j_0+2} & \dots & a_{2,j_0+L-1} & a_{2,j_0+L} & \dots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{i_0+1,1} & a_{i_0+1,2} & \dots & a_{i_0+1,j_0+1} & a_{i_0+1,j_0+2} & \dots & a_{i_0+1,j_0+L-1} & a_{i_0+1,j_0+L} & \dots & a_{i_0+1,N} \\ a_{i_0+2,1} & a_{i_0+2,2} & \dots & a_{i_0+2,j_0+1} & a_{i_0+2,j_0+2} & \dots & a_{i_0+2,j_0+L-1} & a_{i_0+2,j_0+L} & \dots & a_{i_0+2,N} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{i_0+K-1,1} & a_{i_0+K-1,2} & \dots & a_{i_0+K-1,j_0+1} & a_{i_0+K-1,j_0+2} & \dots & a_{i_0+K-1,j_0+L-1} & a_{i_0+K-1,j_0+L} & \dots & a_{i_0+K-1,N} \\ a_{i_0+K,1} & a_{i_0+K,2} & \dots & a_{i_0+K,j_0+1} & a_{i_0+K,j_0+2} & \dots & a_{i_0+K,j_0+L-1} & a_{i_0+K,j_0+L} & \dots & a_{i_0+K,N} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{M,1} & a_{M,2} & \dots & a_{M,j_0+1} & a_{M,j_0+2} & \dots & a_{M,j_0+L-1} & a_{M,j_0+L} & \dots & a_{M,N} \end{bmatrix}$$

Column Major Layout

In column major layout the first index, i , of matrix elements $a_{i,j}$ changes faster than the second index when accessing sequential memory locations. In other words, for $1 \leq j < M$, if the element $a_{i,j}$ is stored in a specific location in memory, the element $a_{i+1,j}$ is stored in the next location, and, for $1 \leq j < N$, the element $a_{M,j}$ is stored in the location previous to element $a_{1,j+1}$. So the matrix elements are located in memory according to this sequence:

$$\{a_{1,1}, a_{2,1} \dots a_{M,1}, a_{1,2}, a_{2,2} \dots a_{M,2} \dots \dots a_{1,N}, a_{2,N} \dots a_{M,N}\}$$

Row Major Layout

In row major layout the second index, j , of matrix elements $a_{i,j}$ changes faster than the first index when accessing sequential memory locations. In other words, for $1 \leq j < N$, if the element $a_{i,j}$ is stored in a specific location in memory, the element $a_{i,j+1}$ is stored in the next location, and, for $1 \leq i < M$, the element $a_{i,N}$ is stored in the location previous to element $a_{i+1,1}$. So the matrix elements are located in memory according to this sequence:

$$\{a_{1,1}, a_{1,2} \dots a_{1,N}, a_{2,1}, a_{2,2} \dots a_{2,N} \dots \dots a_{M,1}, a_{M,2} \dots a_{M,N}\}$$

Leading Dimension Parameter

A leading dimension parameter allows use of LAPACK routines on a submatrix of a larger matrix. For example, the submatrix B can be extracted from the original matrix A defined previously:

$$B = \begin{bmatrix} a_{i_0+1,j_0+1} & a_{i_0+1,j_0+2} & \dots & a_{i_0+1,j_0+L-1} & a_{i_0+1,j_0+L} \\ a_{i_0+2,j_0+1} & a_{i_0+2,j_0+2} & \dots & a_{i_0+2,j_0+L-1} & a_{i_0+2,j_0+L} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{i_0+K-1,j_0+1} & a_{i_0+K-1,j_0+2} & \dots & a_{i_0+K-1,j_0+L-1} & a_{i_0+K-1,j_0+L} \\ a_{i_0+K,j_0+1} & a_{i_0+K,j_0+2} & \dots & a_{i_0+K,j_0+L-1} & a_{i_0+K,j_0+L} \end{bmatrix}$$

B is formed from rows with indices $i_0 + 1$ to $i_0 + K$ and columns $j_0 + 1$ to $j_0 + L$ of matrix A . To specify matrix B , LAPACK routines require four parameters:

- the number of rows K ;
- the number of columns L ;
- a pointer to the start of the array containing elements of B ;

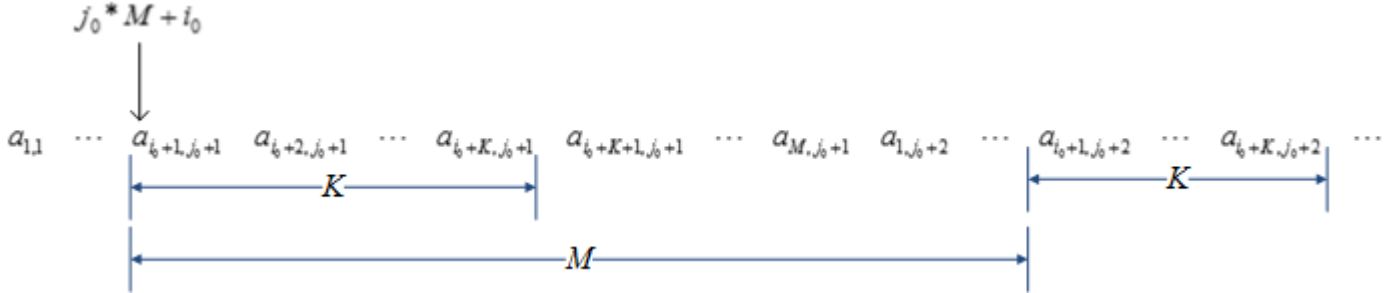
- the leading dimension of the array containing elements of B .

The leading dimension depends on the layout of the matrix:

- Column major layout

Leading dimension $ldb=M$, the number of rows of matrix A .

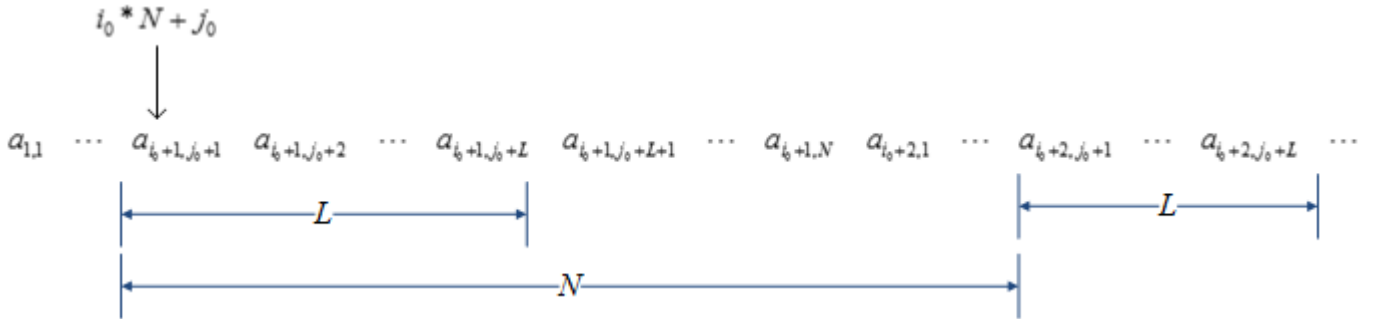
Starting address: offset by $i_0 + j_0 * ldb$ from $a_{1,1}$.



- Row major layout

Leading dimension $ldb=N$, the number of columns of matrix A .

Starting address: offset by $i_0 * ldb + j_0$ from $a_{1,1}$.



Matrix Storage Schemes for LAPACK Routines

LAPACK routines use the following matrix storage schemes:

- Full Storage
- Packed Storage
- Band Storage
- Rectangular Full Packed (RFP) Storage

Full Storage

Consider an m -by- n matrix A :

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & a_{m,3} & \cdots & a_{m,n} \end{pmatrix}$$

It is stored in a one-dimensional array a of length at least $lda*n$ for column major layout or $m*lda$ for row major layout. Element a_{ij} is stored as array element $a[k]$ where the mapping of $k(i, j)$ is defined as

- column major layout: $k(i, j) = i - 1 + (j - 1) * lda$
- row major layout: $k(i, j) = (i - 1) * lda + j - 1$

NOTE

Although LAPACK accepts parameter values of zero for matrix size, in general the size of the array used to store an m -by- n matrix A with leading dimension lda should be greater than or equal to $\max(1, n * lda)$ for column major layout and $\max(1, m * lda)$ for row major layout.

NOTE

Even though the array used to store a matrix is one-dimensional, for simplicity the documentation sometimes refers parts of the array such as rows, columns, upper and lower triangular part, and diagonals. These refer to the parts of the matrix stored within the array. For example, the lower triangle of array a is defined as the subset of elements $a[k(i, j)]$ with $i \geq j$.

Packed Storage

The packed storage format compactly stores matrix elements when only one part of the matrix, the upper or lower triangle, is necessary to determine all of the elements of the matrix. This is the case when the matrix is upper triangular, lower triangular, symmetric, or Hermitian. For an n -by- n matrix of one of these types, a linear array ap of length $n * (n + 1) / 2$ is adequate. Two parameters define the storage scheme:

matrix_layout, which specifies column major (with the value `LAPACK_COL_MAJOR`) or row major (with the value `LAPACK_ROW_MAJOR`) matrix layout, and *uplo*, which specifies that the upper triangle (with the value 'U') or the lower triangle (with the value 'L') is stored.

Element a_{ij} is stored as array element $a[k]$ where the mapping of $k(i, j)$ is defined as

		<i>matrix_layout</i> = <code>LAPACK_COL_MAJOR</code>	<i>matrix_layout</i> = <code>LAPACK_ROW_MAJOR</code>
<i>uplo</i> = 'U'	$1 \leq i \leq j \leq n$	$k(i, j) = i - 1 + j * (j - 1) / 2$	$k(i, j) = j - 1 + (i - 1) * (2 * n - i) / 2$
<i>uplo</i> = 'L'	$1 \leq j \leq i \leq n$	$k(i, j) = i - 1 + (j - 1) * (2 * n - j) / 2$	$k(i, j) = j - 1 + i * (i - 1) / 2$

NOTE

Although LAPACK accepts parameter values of zero for matrix size, in general the size of the array should be greater than or equal to $\max(1, n * (n + 1) / 2)$.

Band Storage

When the non-zero elements of a matrix are confined to diagonal bands, it is possible to store the elements more efficiently using band storage. For example, consider an m -by- n band matrix A with kl subdiagonals and ku superdiagonals:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,k_u+1} & & & \\ \vdots & \vdots & \ddots & \ddots & \ddots & & \\ a_{k_l+1,1} & a_{k_l+1,2} & \cdots & a_{k_l+1,k_u+1} & & a_{k_l+1,k_l+k_u+1} & \\ & a_{k_l+2,2} & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & & a_{k_l+j,j} & a_{k_l+1,j+1} & \cdots & \cdots \cdots a_{k_l+j,k_l+k_u+j} \\ & & & & \ddots & \ddots & \ddots \ddots \ddots \ddots \end{pmatrix}$$

This matrix can be stored compactly in a one dimensional array `ab`. There are two operations involved in storing the matrix: packing the band matrix into matrix AB , and converting the packed matrix to a one-dimensional array.

- Packing the Band Matrix: How the band matrix is packed depends on the matrix layout.
 - Column major layout: matrix A is packed in an $ldab$ -by- n matrix AB column-wise so that the diagonals of A become rows of array AB .

$$AB = \begin{pmatrix} & & & & a_{1,k_u+1} & a_{1,k_u+2} & a_{1,k_u+3} & \cdots \\ & & & \ddots & \vdots & \vdots & \vdots & \cdots \\ & & a_{1,3} & \cdots & a_{k_u-1,k_u+1} & a_{k_u,k_u+2} & a_{k_u+1,k_u+3} & \cdots \\ & a_{1,2} & a_{2,3} & \cdots & a_{k_u,k_u+1} & a_{k_u+1,k_u+2} & a_{k_u+2,k_u+3} & \cdots \\ a_{1,1} & a_{2,2} & a_{3,3} & \cdots & a_{k_u+1,k_u+1} & a_{k_u+2,k_u+2} & a_{k_u+3,k_u+3} & \cdots \\ a_{2,1} & a_{3,2} & a_{4,3} & \cdots & a_{k_u+2,k_u+1} & a_{k_u+3,k_u+2} & a_{k_u+4,k_u+3} & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \cdots \\ a_{k_l+1,1} & a_{k_l+2,2} & a_{k_l+3,3} & \cdots & a_{k_u+k_l+1,k_u+1} & a_{k_u+k_l+2,k_u+2} & a_{k_u+k_l+3,k_u+3} & \cdots \end{pmatrix}$$

The number of rows of AB is $ldab \geq kl + ku + 1$, and the number of columns of AB is n .

- Row major layout: matrix A is packed in an m -by- $ldab$ matrix AB row-wise so that the diagonals of A become columns of AB .

$$AB = \begin{pmatrix} & & & & a_{1,1} & a_{1,2} & \cdots & a_{1,k_u+1} \\ & & & & a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,k_u+2} \\ & & & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & \cdots & a_{3,k_u+3} \\ & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{k_l+1,1} & a_{k_l+1,2} & \cdots & \cdots & a_{k_l+1,k_l+1} & a_{k_l+1,k_l+2} & \cdots & a_{k_l+1,k_u+k_l+1} \\ a_{k_l+2,2} & a_{k_l+2,3} & \cdots & \cdots & a_{k_l+2,k_l+2} & a_{k_l+2,k_l+3} & \cdots & a_{k_l+2,k_u+k_l+2} \\ a_{k_l+3,3} & a_{k_l+3,4} & \cdots & \cdots & a_{k_l+3,k_l+3} & a_{k_l+3,k_l+4} & \cdots & a_{k_l+3,k_u+k_l+3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

The number of columns of AB is $ldab \geq kl + ku + 1$, and the number of rows of AB is m .

NOTE

For both column major and row major layout, elements of the upper left triangle of AB are not used. Depending on the relationship of the dimensions m , n , kl , and ku , the lower right triangle might not be used.

- Converting the Packed Matrix to a One-Dimensional Array: The packed matrix AB is stored in a linear array ab as described in [Full Storage](#) . The size of ab should be greater than or equal to the total number of elements of matrix AB : $ldab*n$ for column major layout or $ldab*m$ for row major layout. The leading dimension of ab , $ldab$, must be greater than or equal to $kl + ku + 1$ (and some routines require it to be even larger).

Element a_{ij} is stored as array element $a[k(i, j)]$ where the mapping of $k(i, j)$ is defined as

- column major layout: $k(i, j) = i + ku - j + (j - 1)*ldab$; $1 \leq j \leq n$, $\max(1, j - ku) \leq i \leq \min(m, j + kl)$
- row major layout: $k(i, j) = j - i + kl + (i - 1)(kl + ku + 1)$, $1 \leq i \leq m$, $\max(1, i - kl) \leq j \leq \min(n, i + ku)$

NOTE

Although LAPACK accepts parameter values of zero for matrix size, in general the size of the array should be greater than or equal to $\max(1, n*ldab)$ for column major layout and $\max(1, m*ldab)$ for row major layout.

Rectangular Full Packed Storage

A combination of full and packed storage, rectangular full packed storage can be used to store the upper or lower triangle of a matrix which is upper triangular, lower triangular, symmetric, or Hermitian. It offers the storage savings of packed storage plus the efficiency of using full storage Level 3 BLAS and LAPACK routines. Three parameters define the storage scheme: *matrix_layout*, which specifies column major (with the value `LAPACK_COL_MAJOR`) or row major (with the value `LAPACK_ROW_MAJOR`) matrix layout; *uplo*, which specifies that the upper triangle (with the value 'U') or the lower triangle (with the value 'L') is stored; and *transr*, which specifies normal (with the value 'N'), transpose (with the value 'T'), or conjugate transpose (with the value 'C') operation on the matrix.

Consider an N -by- N matrix A :

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,N-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,N-1} \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots & a_{2,N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N-1,0} & a_{N-1,1} & a_{N-1,2} & \cdots & a_{N-1,N-1} \end{pmatrix}$$

The upper or lower triangle of A can be stored in the array ap of length $N*(N + 1)/2$.

Additionally, define k as the integer part of $N/2$, such that $N=2*k$ if N is even, and $N=2*k + 1$ if N is odd.

Storing the matrix involves packing the matrix into a rectangular matrix, and then storing the matrix in a one-dimensional array. The size of rectangular matrix AP required for the N -by- N matrix A is $N + 1$ by $N/2$ for even N , and N by $(N + 1)/2$ for odd N .

These examples illustrate the rectangular full packed storage method.

- Upper triangular - *uplo* = 'U'

Consider a matrix A with $N = 6$:

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} \end{pmatrix}$$

- Not transposed - $transr = 'N'$

The elements of the upper triangle of A can be packed in a matrix with the dimensions $(N + 1)$ -by- $(N/2) = 7$ by 3:

$$AP = \begin{pmatrix} a_{0,3} & a_{0,4} & a_{0,5} \\ a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,3} & a_{2,4} & a_{2,5} \\ a_{3,3} & a_{3,4} & a_{3,5} \\ a_{0,0} & a_{4,4} & a_{4,5} \\ a_{0,1} & a_{1,1} & a_{5,5} \\ a_{0,2} & a_{1,2} & a_{2,2} \end{pmatrix}$$

- Transposed or conjugate transposed - $transr = 'T'$ or $transr = 'C'$

The elements of the upper triangle of A can be packed in a matrix with the dimensions $(N/2)$ by $(N + 1) = 3$ by 7:

$$AP = \begin{pmatrix} a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} & a_{0,0} & a_{0,1} & a_{0,2} \\ a_{0,4} & a_{1,4} & a_{2,4} & a_{3,4} & a_{4,4} & a_{1,1} & a_{1,2} \\ a_{0,5} & a_{1,5} & a_{2,5} & a_{3,5} & a_{4,5} & a_{5,5} & a_{2,2} \end{pmatrix}$$

Consider a matrix A with $N = 5$:

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}$$

- Not transposed - $transr = 'N'$

The elements of the upper triangle of A can be packed in a matrix with the dimensions (N) -by- $((N + 1)/2) = 5$ by 3:

$$AP = \begin{pmatrix} a_{0,2} & a_{0,3} & a_{0,4} \\ a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,2} & a_{2,3} & a_{2,4} \\ a_{0,0} & a_{3,3} & a_{3,4} \\ a_{0,1} & a_{1,1} & a_{4,4} \end{pmatrix}$$

- Transposed or conjugate transposed - $transr = 'T'$ or $transr = 'C'$

The elements of the upper triangle of A can be packed in a matrix with the dimensions $((N+1)/2)$ by $(N) = 3$ by 5:

$$AP = \begin{pmatrix} a_{0,2} & a_{1,2} & a_{2,3} & a_{0,0} & a_{0,1} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} & a_{1,1} \\ a_{0,4} & a_{1,4} & a_{2,4} & a_{3,4} & a_{4,4} \end{pmatrix}$$

- Lower triangular - $uplo = 'L'$

Consider a matrix A with $N = 6$:

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} \end{pmatrix}$$

- Not transposed - $transr = 'N'$

The elements of the lower triangle of A can be packed in a matrix with the dimensions $(N + 1)$ -by- $(N/2) = 7$ by 3:

$$AP = \begin{pmatrix} a_{3,3} & a_{4,3} & a_{5,3} \\ a_{0,0} & a_{4,4} & a_{5,4} \\ a_{1,0} & a_{1,1} & a_{5,5} \\ a_{2,0} & a_{2,1} & a_{2,2} \\ a_{3,0} & a_{3,1} & a_{3,2} \\ a_{3,0} & a_{4,1} & a_{4,2} \\ a_{5,0} & a_{5,1} & a_{5,2} \end{pmatrix}$$

- Transposed or conjugate transposed - $transr = 'T'$ or $transr = 'C'$

The elements of the lower triangle of A can be packed in a matrix with the dimensions $(N/2)$ by $(N + 1) = 3$ by 7:

$$AP = \begin{pmatrix} a_{3,3} & a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} & a_{4,0} & a_{5,0} \\ a_{4,3} & a_{4,4} & a_{1,1} & a_{2,1} & a_{3,1} & a_{4,1} & a_{5,1} \\ a_{5,3} & a_{5,4} & a_{5,5} & a_{2,2} & a_{3,2} & a_{4,2} & a_{5,2} \end{pmatrix}$$

Consider a matrix A with $N = 5$:

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}$$

- Not transposed - $transr = 'N'$

The elements of the lower triangle of A can be packed in a matrix with the dimensions (N) -by- $((N + 1)/2) = 5$ by 3:

$$AP = \begin{pmatrix} a_{0,0} & a_{3,3} & a_{4,3} \\ a_{1,0} & a_{1,1} & a_{4,4} \\ a_{2,0} & a_{2,1} & a_{2,2} \\ a_{3,0} & a_{3,1} & a_{3,2} \\ a_{4,0} & a_{4,1} & a_{4,2} \end{pmatrix}$$

- Transposed or conjugate transposed - $transr = 'T'$ or $transr = 'C'$

The elements of the lower triangle of A can be packed in a matrix with the dimensions $((N+1)/2)$ by $(N) = 3$ by 5:

$$AP = \begin{pmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} & a_{4,0} \\ a_{3,3} & a_{1,1} & a_{2,1} & a_{3,1} & a_{4,1} \\ a_{4,3} & a_{4,4} & a_{2,2} & a_{3,2} & a_{4,2} \end{pmatrix}$$

The packed matrix AP can be stored using column major layout or row major layout.

NOTE

The *matrix_layout* and *transr* parameters can specify the same storage scheme: for example, the storage scheme for *matrix_layout* = LAPACK_COL_MAJOR and *transr* = 'N' is the same as that for *matrix_layout* = LAPACK_ROW_MAJOR and *transr* = 'T'.

Element a_{ij} is stored as array element $ap[l]$ where the mapping of $l(i, j)$ is defined in the following tables.

- Column major layout: *matrix_layout* = LAPACK_COL_MAJOR a

<i>transr</i>	<i>uplo</i>	N	$l(i, j) =$	i	j
'N'	'U'	$2*k$	$(j - k)*(N + 1) + i$	$0 \leq i < N$	$\max(i, k) \leq j < N$
			$i*(N + 1) + j + k + 1$	$0 \leq i < k$	$i \leq j < k$
		$2*k + 1$	$(j - k)*N + i$	$0 \leq i < N$	$\max(i, k) \leq j < N$
			$i*N + j + k + 1$	$0 \leq i < k$	$i \leq j < k$
	'L'	$2*k$	$j*(N + 1) + i + 1$	$0 \leq i < N$	$0 \leq j \leq \min(i, k)$
			$(i - k)*(N + 1) + j - k$	$k \leq i < N$	$k \leq j \leq i$
		$2*k + 1$	$j*N + i$	$0 \leq i < N$	$0 \leq j \leq \min(i, k)$
			$(i - k)*N + j - k - 1$	$k + 1 \leq i < N$	$k + 1 \leq j \leq i$
'T' or 'C'	'U'	$2*k$	$i*k + j - k$	$0 \leq i < N$	$\max(i, k) \leq j < N$
			$(j + k + 1)*k + i$	$0 \leq i < k$	$i \leq j < k$
		$2*k + 1$	$i*(k + 1) + j - k$	$0 \leq i < N$	$\max(i, k) \leq j < N$
			$(j + k + 1)*(k + 1) + i$	$0 \leq i < k$	$i \leq j < k$
	'L'	$2*k$	$(i + 1)*k + j$	$0 \leq i < N$	$0 \leq j \leq \min(i, k)$
			$(j - k)*k + i - k$	$k \leq i < N$	$k \leq j \leq i$

<i>trans_r</i>	<i>uplo</i>	<i>N</i>	<i>l(i, j) =</i>	<i>i</i>	<i>j</i>
		$2*k + 1$	$i*(k + 1) + j$	$0 \leq i < N$	$0 \leq j \leq \min(i, k)$
			$(j - k - 1)*(k + 1) + i - k$	$k + 1 \leq i < N$	$k + 1 \leq j \leq i$

- Row major layout: *matrix_layout* = LAPACK_ROW_MAJOR

<i>trans_r</i>	<i>uplo</i>	<i>N</i>	<i>l(i, j) =</i>	<i>i</i>	<i>j</i>
'N'	'U'	$2*k$	$i*k + j - k$	$0 \leq i < N$	$\max(i, k) \leq j < N$
			$(k + j + 1)*k + i$	$0 \leq i < k$	$i \leq j < k$
		$2*k + 1$	$i*(k + 1) + j - k$	$0 \leq i < N$	$\max(i, k) \leq j < N$
			$(k + j + 1)*(k + 1) + i$	$0 \leq i < k$	$i \leq j < k$
	'L'	$2*k$	$(i + 1)*k + j$	$0 \leq i < N$	$0 \leq j \leq \min(i, k)$
			$(j - k)*k + i - k$	$k \leq i < N$	$k \leq j \leq i$
		$2*k + 1$	$i*(k + 1) + j$	$0 \leq i < N$	$0 \leq j \leq \min(i, k)$
			$(j - k - 1)*(k + 1) + i - k$	$k + 1 \leq i < N$	$k + 1 \leq j \leq i$
'T' or 'C'	'U'	$2*k$	$(j - k)*(N + 1) + i$	$0 \leq i < N$	$\max(i, k) \leq j < N$
			$i*(N + 1) + k + j + 1$	$0 \leq i < k$	$i \leq j < k$
		$2*k + 1$	$(j - k)*N + i$	$0 \leq i < N$	$\max(i, k) \leq j < N$
			$i*N + k + j + 1$	$0 \leq i < k$	$i \leq j < k$
	'L'	$2*k$	$j*(N + 1) + i + 1$	$0 \leq i < N$	$0 \leq j \leq \min(i, k)$
			$(i - k)*(N + 1) + j - k$	$k \leq i < N$	$k \leq j \leq i$
		$2*k + 1$	$j*N + i$	$0 \leq i < N$	$0 \leq j \leq \min(i, k)$
			$(i - k)*N + j - k - 1$	$k + 1 \leq i < N$	$k + 1 \leq j \leq i$

NOTE

Although LAPACK accepts parameter values of zero for matrix size, in general the size of the array should be greater than or equal to $\max(1, N*(N + 1)/2)$.

Mathematical Notation for LAPACK Routines

Descriptions of LAPACK routines use the following notation:

A^H For an M -by- N matrix A , denotes the conjugate transposed N -by- M matrix with elements:

$$a_{i,j}^H = \overline{a_{j,i}}$$

For a real-valued matrix, $A^H = A^T$.

$x \cdot y$ The *dot product* of two vectors, defined as:

$$x \cdot y = \sum_i x_i \overline{y_i}$$

$Ax = b$ A system of linear equations with an n -by- n matrix $A = \{a_{ij}\}$, a right-hand side vector $b = \{b_i\}$, and an unknown vector $x = \{x_i\}$.

$AX = B$ A set of systems with a common matrix A and multiple right-hand sides. The columns of B are individual right-hand sides, and the columns of X are the corresponding solutions.

$|x|$ the vector with elements $|x_i|$ (absolute values of x_i).

$|A|$ the matrix with elements $|a_{ij}|$ (absolute values of a_{ij}).

$||x||_\infty = \max_i |x_i|$ The *infinity-norm* of the vector x .

$||A||_\infty = \max_i \sum_j |a_{ij}|$ The *infinity-norm* of the matrix A .

$||A||_1 = \max_j \sum_i |a_{ij}|$ The *one-norm* of the matrix A . $||A||_1 = ||A^T||_\infty = ||A^H||_\infty$

$||x||_2$ The *2-norm* of the vector x : $||x||_2 = (\sum_i |x_i|^2)^{1/2} = ||x||_E$ (see the definition for *Euclidean norm* in this topic).

$||A||_2$ The *2-norm* (or *spectral norm*) of the matrix A .

$$||A||_2 = \max_i \sigma_i, ||A||_2^2 = \max_{||x||_E=1} (Ax \cdot Ax)$$

$||A||_E$ The *Euclidean norm* of the matrix A : $||A||_E^2 = \sum_i \sum_j |a_{ij}|^2$.

$\kappa(A) = ||A|| \cdot ||A^{-1}||$ The *condition number* of the matrix A .

λ_i *Eigenvalues* of the matrix A (for the definition of eigenvalues, see [Eigenvalue Problems](#)).

σ_i *Singular values* of the matrix A . They are equal to square roots of the eigenvalues of $A^H A$. (For more information, see [Singular Value Decomposition](#)).

Error Analysis

In practice, most computations are performed with rounding errors. Besides, you often need to solve a system $Ax = b$, where the data (the elements of A and b) are not known exactly. Therefore, it is important to understand how the data errors and rounding errors can affect the solution x .

Data perturbations. If x is the exact solution of $Ax = b$, and $x + \delta x$ is the exact solution of a perturbed problem $(A + \delta A)(x + \delta x) = (b + \delta b)$, then this estimate, given up to linear terms of perturbations, holds:

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right)$$

where $A + \delta A$ is nonsingular and

$$\kappa(A) = \|A\| \|A^{-1}\|.$$

In other words, relative errors in A or b may be amplified in the solution vector x by a factor $\kappa(A) = \|A\| \|A^{-1}\|$ called the *condition number* of A .

Rounding errors have the same effect as relative perturbations $c(n)\varepsilon$ in the original data. Here ε is the *machine precision*, defined as the smallest positive number x such that $1 + x > 1$; and $c(n)$ is a modest function of the matrix order n . The corresponding solution error is

$\|\delta x\| / \|x\| \leq c(n) \kappa(A) \varepsilon$. (The value of $c(n)$ is seldom greater than $10n$.)

NOTE

Machine precision depends on the data type used. For example, it is usually defined in the `float.h` file as `FLT_EPSILON` the `float` datatype and `DBL_EPSILON` for the `double` datatype.

Thus, if your matrix A is *ill-conditioned* (that is, its condition number $\kappa(A)$ is very large), then the error in the solution x can also be large; you might even encounter a complete loss of precision. LAPACK provides routines that allow you to estimate $\kappa(A)$ (see [Routines for Estimating the Condition Number](#)) and also give you a more precise estimate for the actual solution error (see [Refining the Solution and Estimating Its Error](#)).

LAPACK Linear Equation Routines

This section describes routines for performing the following computations:

- factoring the matrix (except for triangular matrices)
- equilibrating the matrix (except for RFP matrices)
- solving a system of linear equations
- estimating the condition number of a matrix (except for RFP matrices)

- refining the solution of linear equations and computing its error bounds (except for RFP matrices)
- inverting the matrix.

To solve a particular problem, you can call two or more [computational routines](#) or call a corresponding [driver routine](#) that combines several tasks in one call. For example, to solve a system of linear equations with a general matrix, call `?getrf` (*LU* factorization) and then `?getrs` (computing the solution). Then, call `?gerfs` to refine the solution and get the error bounds. Alternatively, use the driver routine `?gesvx` that performs all these tasks in one call.

LAPACK Linear Equation Computational Routines

Table "Computational Routines for Systems of Equations with Real Matrices" lists the LAPACK computational routines for factorizing, equilibrating, and inverting *real* matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error. Table "Computational Routines for Systems of Equations with Complex Matrices" lists similar routines for *complex* matrices.

Computational Routines for Systems of Equations with Real Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	<code>?getrf</code>	<code>?geequ</code> , <code>?geequb</code>	<code>?getrs</code>	<code>?gecon</code>	<code>?gerfs</code> , <code>?gerfsx</code>	<code>?getri</code>
general band	<code>?gbtrf</code>	<code>?gbequ</code> , <code>?gbequb</code>	<code>?gbtrs</code>	<code>?gbcon</code>	<code>?gbrfs</code> , <code>?gbrfsx</code>	
general tridiagonal	<code>?gttrf</code>		<code>?gttrs</code>	<code>?gtcon</code>	<code>?gtrfs</code>	
diagonally dominant tridiagonal	<code>?dttrfb</code>		<code>?dttrs</code>			
symmetric positive-definite	<code>?potrf</code>	<code>?poequ</code> , <code>?poequb</code>	<code>?potrs</code>	<code>?pocon</code>	<code>?porfs</code> , <code>?porfsx</code>	<code>?potri</code>
symmetric positive-definite, packed storage	<code>?pptrf</code>	<code>?ppequ</code>	<code>?pptrs</code>	<code>?ppcon</code>	<code>?pprfs</code>	<code>?pptri</code>
symmetric positive-definite, RFP storage	<code>?pftrf</code>		<code>?pftrs</code>			<code>?pftri</code>
symmetric positive-definite, band	<code>?pbtrf</code>	<code>?pbequ</code>	<code>?pbtrs</code>	<code>?pbcon</code>	<code>?pbrfs</code>	
symmetric positive-definite, tridiagonal	<code>?pttrf</code>		<code>?pttrs</code>	<code>?ptcon</code>	<code>?ptrfs</code>	
symmetric indefinite	<code>?sytrf</code> <code>?sytrf_rk</code> <code>?sytrf_aa</code>	<code>?syequb</code>	<code>?sytrs</code> <code>?sytrs2</code> <code>?sytrs3</code> <code>?sytrs_aa</code>	<code>?sycon</code> <code>?sycon_3</code>	<code>?sytrfs</code> , <code>?sytrfsx</code>	<code>?sytri</code> <code>?sytri2</code> <code>?sytri2x</code> <code>?sytri_3</code>
symmetric indefinite, packed storage	<code>?sptrf</code> <code>mkl_?spffrt2</code> , <code>mkl_?spffrtx</code>		<code>?sptrs</code>	<code>?spcon</code>	<code>?sprfs</code>	<code>?sptri</code>

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
triangular			?trtrs	?trcon	?trrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tprfs	?tptri
triangular, RFP storage						?tftri
triangular band			?tbtrs	?tbcon	?tbrfs	

Computational Routines for Systems of Equations with Complex Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	?getrf	?geequ, ?geequb	?getrs	?gecon	?gerfs, ?gerfsx	?getri
general band	?gbtrf	?gbequ, ?gbequb	?gbtrs	?gbcon	?gbrfs, ?gbrfsx	
general tridiagonal	?gttrf		?gttrs	?gtcon	?gtrfs	
Hermitian positive-definite	?potrf	?poequ, ?poequb	?potrs	?pocon	?porfs, ?porfsx	?potri
Hermitian positive-definite, packed storage	?pptrf	?ppequ	?pptrs	?ppcon	?pprfs	?pptri
Hermitian positive-definite, RFP storage	?pftrf		?pftrs			?pftri
Hermitian positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	
Hermitian positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
Hermitian indefinite	?hetrf ?hetrf_rk ?hetrf_aa	?heequb	?hetrs ?hetrs2 ?hetrs_3 ?hetrs_aa	?hecon ?hecon_3	?herfs, ?herfsx	?hetri ?hetri2 ?hetri2x ?hetri_3
symmetric indefinite	?sytrf ?sytrf_rk	?syequb	?sytrs ?sytrs2 ?sytrs3	?sycon ?sycon_3	?syarfs, ?syarfsx	?sytri ?sytri2 ?sytri2x ?sytri_3
Hermitian indefinite, packed storage	?hptrf		?hptrs	?hpcon	?hprfs	?hptri

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
symmetric indefinite, packed storage	<code>?spturf</code> <code>mkl_?spffrt2, mkl_?spffrtx</code>		<code>?spttrs</code>	<code>?spcon</code>	<code>?sprfs</code>	<code>?sptri</code>
triangular			<code>?trtrs</code>	<code>?trcon</code>	<code>?trrfs</code>	<code>?trtri</code>
triangular, packed storage			<code>?tptrs</code>	<code>?tpcon</code>	<code>?tprfs</code>	<code>?tptri</code>
triangular, RFP storage						<code>?tftri</code>
triangular band			<code>?tbtrs</code>	<code>?tbcon</code>	<code>?tbrfs</code>	

Matrix Factorization: LAPACK Computational Routines

This section describes the LAPACK routines for matrix factorization. The following factorizations are supported:

- *LU* factorization
- Cholesky factorization of real symmetric positive-definite matrices
- Cholesky factorization of real symmetric positive-definite matrices with pivoting
- Cholesky factorization of Hermitian positive-definite matrices
- Cholesky factorization of Hermitian positive-definite matrices with pivoting
- Bunch-Kaufman factorization of real and complex symmetric matrices
- Bunch-Kaufman factorization of Hermitian matrices.

You can compute:

- the *LU* factorization using full and band storage of matrices
- the Cholesky factorization using full, packed, RFP, and band storage
- the Bunch-Kaufman factorization using full and packed storage.

`?getrf`

Computes the LU factorization of a general m-by-n matrix.

Syntax

```
lapack_int LAPACKESgetrf (int matrix_layout , lapack_int m , lapack_int n , float *
a , lapack_int lda , lapack_int * ipiv );

lapack_int LAPACKEDgetrf (int matrix_layout , lapack_int m , lapack_int n , double *
a , lapack_int lda , lapack_int * ipiv );

lapack_int LAPACKECgetrf (int matrix_layout , lapack_int m , lapack_int n ,
lapack_complex_float * a , lapack_int lda , lapack_int * ipiv );

lapack_int LAPACKEZgetrf (int matrix_layout , lapack_int m , lapack_int n ,
lapack_complex_double * a , lapack_int lda , lapack_int * ipiv );
```

Include Files

- `mkl.h`

Description

The routine computes the LU factorization of a general m -by- n matrix A as

$$A = P * L * U,$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	The number of columns in A ; $n \geq 0$.
<i>a</i>	Array, size at least $\max(1, lda * n)$ for column-major layout or $\max(1, lda * m)$ for row-major layout. Contains the matrix A .
<i>lda</i>	The leading dimension of array a , which must be at least $\max(1, m)$ for column-major layout or $\max(1, n)$ for row-major layout.

Output Parameters

<i>a</i>	Overwritten by L and U . The unit diagonal elements of L are not stored.
<i>ipiv</i>	Array, size at least $\max(1, \min(m, n))$. Contains the pivot indices; for $1 \leq i \leq \min(m, n)$, row i was interchanged with row $ipiv(i)$.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

Application Notes

The computed L and U are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(\min(m, n)) \varepsilon P |L| |U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$$(2/3)n^3 \quad \text{If } m = n,$$

$$(1/3)n^2(3m-n) \quad \text{If } m > n,$$

$(1/3)m^2(3n-m)$ If $m < n$.

The number of operations for complex flavors is four times greater.

After calling this routine with $m = n$, you can call the following:

<code>?getrs</code>	to solve $A * X = B$ or $A^T X = B$ or $A^H X = B$
<code>?gecon</code>	to estimate the condition number of A
<code>?getri</code>	to compute the inverse of A .

See Also

[mkl_progress](#)

Matrix Storage Schemes

mkl_?getrfnp

Computes the LU factorization of a general m-by-n matrix without pivoting.

Syntax

```
lapack_int LAPACKE_mkl_sgetrfnp (int matrix_layout , lapack_int m , lapack_int n ,
float * a , lapack_int lda );
```

```
lapack_int LAPACKE_mkl_dgetrfnp (int matrix_layout , lapack_int m , lapack_int n ,
double * a , lapack_int lda );
```

```
lapack_int LAPACKE_mkl_cgetrfnp (int matrix_layout , lapack_int m , lapack_int n ,
lapack_complex_float * a , lapack_int lda );
```

```
lapack_int LAPACKE_mkl_zgetrfnp (int matrix_layout , lapack_int m , lapack_int n ,
lapack_complex_double * a , lapack_int lda );
```

Include Files

- `mkl.h`

Description

The routine computes the *LU* factorization of a general *m*-by-*n* matrix *A* as

$$A = L * U,$$

where *L* is lower triangular with unit-diagonal elements (lower trapezoidal if $m > n$) and *U* is upper triangular (upper trapezoidal if $m < n$). The routine does not use pivoting.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	The number of columns in <i>A</i> ; $n \geq 0$.
<i>a</i>	Array, size at least $\max(1, lda * n)$ for column-major layout or $\max(1, lda * m)$ for row-major layout. Contains the matrix <i>A</i> .

lda The leading dimension of array *a*, which must be at least $\max(1, m)$ for column-major layout or $\max(1, n)$ for row-major layout.

Output Parameters

a Overwritten by *L* and *U*. The unit diagonal elements of *L* are not stored.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The approximate number of floating-point operations for real flavors is

$(2/3)n^3$ If $m = n$,

$(1/3)n^2(3m-n)$ If $m > n$,

$(1/3)m^2(3n-m)$ If $m < n$.

The number of operations for complex flavors is four times greater.

After calling this routine with $m = n$, you can call the following:

`mkl_?getrinp` to compute the inverse of *A*

See Also

`mkl_progress`

Matrix Storage Schemes

`mkl_?getrfnpi`

Performs LU factorization (complete or incomplete) of a general matrix without pivoting.

Syntax

```
lapack_int LAPACK_mkl_sgetrfnpi (int matrix_layout, lapack_int m, lapack_int n,
lapack_int nfact, float* a, lapack_int lda);
```

```
lapack_int LAPACK_mkl_dgetrfnpi (int matrix_layout, lapack_int m, lapack_int n,
lapack_int nfact, double* a, lapack_int lda);
```

```
lapack_int LAPACK_mkl_cgetrfnpi (int matrix_layout, lapack_int m, lapack_int n,
lapack_int nfact, lapack_complex_float* a, lapack_int lda);
```

```
lapack_int LAPACK_mkl_zgetrfnpi (int matrix_layout, lapack_int m, lapack_int n,
lapack_int nfact, lapack_complex_double* a, lapack_int lda);
```

Include Files

- `mk1.h`

Description

The routine computes the LU factorization of a general m -by- n matrix A without using pivoting. It supports incomplete factorization. The factorization has the form:

$$A = L * U,$$

where L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).

Incomplete factorization has the form:

$$A = L * U + \tilde{A}$$

where L is lower trapezoidal with unit diagonal elements, U is upper trapezoidal, and

$$\tilde{A}$$

is the unfactored part of matrix A . See the application notes section for further details.

NOTE

Use `?getrf` if it is possible that the matrix is not diagonal dominant.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>m</code>	The number of rows in matrix A ; $m \geq 0$.
<code>n</code>	The number of columns in matrix A ; $n \geq 0$.
<code>nfact</code>	The number of rows and columns to factor; $0 \leq nfact \leq \min(m, n)$. Note that if $nfact < \min(m, n)$, incomplete factorization is performed.
<code>a</code>	Array of size at least $lda * n$ for column major layout and at least $lda * m$ for row major layout. Contains the matrix A .
<code>lda</code>	The leading dimension of array a . $lda \geq \max(1, m)$ for column major layout and $lda \geq \max(1, n)$ for row major layout.

Output Parameters

<code>a</code>	Overwritten by L and U . The unit diagonal elements of L are not stored. When incomplete factorization is specified by setting $nfact < \min(m, n)$, a also contains the unfactored submatrix \tilde{A}_{22} . See the application notes section for further details.
----------------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, u_{ij} is 0. The requested factorization has been completed, but *U* is exactly singular. Division by 0 will occur if factorization is completed and factor *U* is used for solving a system of linear equations.

Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix $A + E$, with

$$|E| \leq c(\min(m, n)) \varepsilon |L| |U|$$

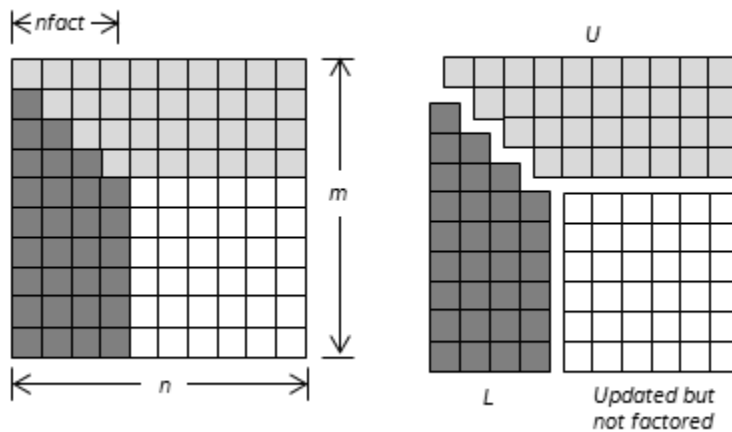
where $c(n)$ is a modest linear function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$(2/3)n^3$	If $m = n = n_{fact}$
$(1/3)n^2(3m-n)$	If $m > n = n_{fact}$
$(1/3)m^2(3n-m)$	If $m = n_{fact} < n$
$(2/3)n^3 - (n-n_{fact})^3$	If $m = n, n_{fact} < \min(m, n)$
$(1/3)(n^2(3m-n) - (n-n_{fact})^2(3m - 2n_{fact} - n))$	If $m > n > n_{fact}$
$(1/3)(m^2(3n-m) - (m-n_{fact})^2(3n - 2n_{fact} - m))$	If $n_{fact} < m < n$

The number of operations for complex flavors is four times greater.

When incomplete factorization is specified, the first *nfact* rows and columns are factored, with the update of the remaining rows and columns of *A* as follows:



If matrix *A* is represented as a block 2-by-2 matrix:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where

- A_{11} is a square matrix of order $nfact$,
- A_{21} is an $(m - nfact)$ -by- $nfact$ matrix,
- A_{12} is an $nfact$ -by- $(n - nfact)$ matrix, and
- A_{22} is an $(m - nfact)$ -by- $(n - nfact)$ matrix.

The result is

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} \cdot \begin{bmatrix} U_1 & U_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & \tilde{A}_{22} \end{bmatrix}$$

L_1 is a lower triangular square matrix of order $nfact$ with unit diagonal and U_1 is an upper triangular square matrix of order $nfact$. L_1 and U_1 result from LU factorization of matrix A_{11} : $A_{11} = L_1 U_1$.

L_2 is an $(m - nfact)$ -by- $nfact$ matrix and $L_2 = A_{21} U_1^{-1}$. U_2 is an $nfact$ -by- $(n - nfact)$ matrix and $U_2 = L_1^{-1} A_{12}$.

$$\tilde{A}_{22}$$

is an $(m - nfact)$ -by- $(n - nfact)$ matrix and

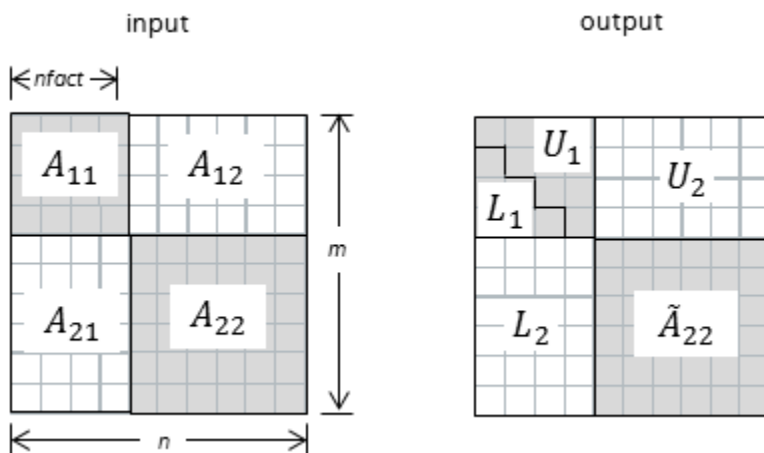
$$\tilde{A}_{22}$$

$$= A_{22} - L_2 U_2.$$

On exit, elements of the upper triangle U_1 are stored in place of the upper triangle of block A_{11} in array a ; elements of the lower triangle L_1 are stored in the lower triangle of block A_{11} in array a (unit diagonal elements are not stored). Elements of L_2 replace elements of A_{21} ; U_2 replaces elements of A_{12} and

$$\tilde{A}_{22}$$

replaces elements of A_{22} .



?getrf2

Computes LU factorization using partial pivoting with row interchanges.

Syntax

```
lapack_int LAPACK_sgetrf2 (int matrix_layout, lapack_int m, lapack_int n, float * a,
lapack_int lda, lapack_int * ipiv);
```

```

lapack_int LAPACKE_dgetrf2 (int matrix_layout, lapack_int m, lapack_int n, double * a,
lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_cgetrf2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float * a, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_zgetrf2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double * a, lapack_int lda, lapack_int * ipiv);

```

Include Files

- mkl.h

Description

?getrf2 computes an LU factorization of a general m -by- n matrix A using partial pivoting with row interchanges.

The factorization has the form

$$A = P * L * U$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and U is upper triangular (upper trapezoidal if $m < n$).

This is the recursive version of the algorithm. It divides the matrix into four submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

where A_{11} is n_1 by n_1 and A_{22} is n_2 by n_2 with $n_1 = \min(m, n)$, and $n_2 = n - n_1$.

The subroutine calls itself to factor $\begin{pmatrix} A_{11} \\ A_{12} \end{pmatrix}$,

do the swaps on $\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}$, solve A_{12} , update A_{22} , then it calls itself to factor A_{22} and do the swaps on A_{21} .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows of the matrix A. $m \geq 0$.
<i>n</i>	The number of columns of the matrix A. $n \geq 0$.
<i>a</i>	Array, size $lda*n$. On entry, the m -by- n matrix to be factored.
<i>lda</i>	The leading dimension of the array <i>a</i> . $lda \geq \max(1, m)$.

Output Parameters

<i>a</i>	On exit, the factors L and U from the factorization $A = P * L * U$; the unit diagonal elements of L are not stored.
<i>ipiv</i>	Array, size $\min(m, n)$. The pivot indices; for $1 \leq i \leq \min(m, n)$, row i of the matrix was interchanged with row $ipiv[i - 1]$.

Return Values

This function returns a value *info*.

= 0: successful exit

< 0: if *info* = $-i$, the i -th argument had an illegal value.

> 0: if *info* = i , $U_{i,i}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?gbtrf

Computes the LU factorization of a general m -by- n band matrix.

Syntax

```
lapack_int LAPACK_E_sgbtrf (int matrix_layout , lapack_int m , lapack_int n , lapack_int
kl , lapack_int ku , float * ab , lapack_int ldab , lapack_int * ipiv );
```

```
lapack_int LAPACK_E_dgbtrf (int matrix_layout , lapack_int m , lapack_int n , lapack_int
kl , lapack_int ku , double * ab , lapack_int ldab , lapack_int * ipiv );
```

```
lapack_int LAPACK_E_cgbtrf (int matrix_layout , lapack_int m , lapack_int n , lapack_int
kl , lapack_int ku , lapack_complex_float * ab , lapack_int ldab , lapack_int * ipiv );
```

```
lapack_int LAPACK_E_zgbtrf (int matrix_layout , lapack_int m , lapack_int n , lapack_int
kl , lapack_int ku , lapack_complex_double * ab , lapack_int ldab , lapack_int *
ipiv );
```

Include Files

- mkl.h

Description

The routine forms the LU factorization of a general m -by- n band matrix A with kl non-zero subdiagonals and ku non-zero superdiagonals, that is,

$$A = P * L * U,$$

where P is a permutation matrix; L is lower triangular with unit diagonal elements and at most kl non-zero elements in each column; U is an upper triangular band matrix with $kl + ku$ superdiagonals. The routine uses partial pivoting, with row interchanges (which creates the additional kl superdiagonals in U).

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows in matrix A ; $m \geq 0$.
<i>n</i>	The number of columns in matrix A ; $n \geq 0$.
<i>kl</i>	The number of subdiagonals within the band of A ; $kl \geq 0$.

<i>ku</i>	The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>ab</i>	Array, size at least $\max(1, ldab \cdot n)$ for column-major layout or $\max(1, ldab \cdot m)$ for row-major layout. The array <i>ab</i> contains the matrix <i>A</i> in band storage as described in Band Storage .
<i>ldab</i>	The leading dimension of the array <i>ab</i> . ($ldab \geq 2 \cdot kl + ku + 1$)

Output Parameters

<i>ab</i>	Overwritten with elements of <i>L</i> and <i>U</i> . <i>U</i> is stored as an upper triangular band matrix with $kl + ku$ superdiagonals, and <i>L</i> is stored as a lower triangular band matrix with kl subdiagonals (diagonal unit values are not stored). Since the output array has more nonzero elements than the initial matrix <i>A</i> , there are limitations on the value of <i>ldab</i> and the placement of elements of <i>A</i> in array <i>ab</i> . See Application Notes below for further details.
<i>ipiv</i>	Array, size at least $\max(1, \min(m, n))$. The pivot indices; for $1 \leq i \leq \min(m, n)$, row <i>i</i> was interchanged with row <i>ipiv</i> (<i>i</i>).

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, *u_{ii}* is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(kl + ku + 1) \varepsilon P |L| |U|$$

$c(k)$ is a modest linear function of *k*, and ε is the machine precision.

The total number of floating-point operations for real flavors varies between approximately $2n(ku+1)kl$ and $2n(kl+ku+1)kl$. The number of operations for complex flavors is four times greater. All these estimates assume that *kl* and *ku* are much less than $\min(m, n)$.

As described in [Band Storage](#), storage of a band matrix can be considered in two steps: packing band matrix elements into a matrix *AB*, then storing the elements in a linear array *ab* using a full storage scheme. The effect of the ?gbtrf routine on matrix *AB* is illustrated by this example, for $m = n = 6$, $kl = 2$, $ku = 1$.

- `matrix_layout = LAPACK_COL_MAJOR`

On entry:	On exit:
-----------	----------

$$AB = \begin{bmatrix} * & * & * & + & + & + \\ * & * & + & + & + & + \\ * & a_{1,2} & a_{2,3} & a_{3,4} & a_{4,5} & a_{5,6} \\ a_{1,1} & a_{2,2} & a_{3,3} & a_{4,4} & a_{5,5} & a_{6,6} \\ a_{2,1} & a_{3,2} & a_{4,3} & a_{5,4} & a_{6,5} & * \\ a_{3,1} & a_{4,2} & a_{5,3} & a_{6,4} & * & * \end{bmatrix} \quad AB = \begin{bmatrix} * & * & * & u_{1,4} & u_{2,5} & u_{3,6} \\ * & * & u_{1,3} & u_{2,4} & u_{3,5} & u_{4,6} \\ * & u_{1,2} & u_{2,3} & u_{3,4} & u_{4,5} & u_{5,6} \\ u_{1,1} & u_{2,2} & u_{3,3} & u_{4,4} & u_{5,5} & u_{6,6} \\ l_{2,1} & l_{3,2} & l_{4,3} & l_{5,4} & l_{6,5} & * \\ l_{3,1} & l_{4,2} & l_{5,3} & l_{6,4} & * & * \end{bmatrix}$$

- `matrix_layout = LAPACK_ROW_MAJOR`

On entry:	On exit:
$AB = \begin{bmatrix} * & * & a_{1,1} & a_{1,2} & + & + \\ * & a_{2,1} & a_{2,2} & a_{2,3} & + & + \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & + & + \\ a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & + & * \\ a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & * & * \\ a_{6,4} & a_{6,5} & a_{6,6} & * & * & * \end{bmatrix}$	$AB = \begin{bmatrix} * & * & u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} \\ * & l_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} & u_{2,5} \\ l_{3,1} & l_{3,2} & u_{3,3} & u_{3,4} & u_{3,5} & u_{3,6} \\ l_{4,2} & l_{4,3} & u_{4,4} & u_{4,5} & u_{4,6} & * \\ l_{5,3} & l_{5,4} & u_{5,5} & u_{5,6} & * & * \\ l_{6,4} & l_{6,5} & u_{6,6} & * & * & * \end{bmatrix}$

Elements marked * are not used; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

After calling this routine with $m = n$, you can call the following routines:

`gbtrs` to solve $A * X = B$ or $A^T * X = B$ or $A^H * X = B$

`gbcon` to estimate the condition number of A .

See Also

`mkl_progress`

Matrix Storage Schemes

`?gttrf`

Computes the LU factorization of a tridiagonal matrix.

Syntax

```
lapack_int LAPACKE_sgtrf (lapack_int n , float * dl , float * d , float * du , float * du2 , lapack_int * ipiv );
```

```
lapack_int LAPACKE_dgtrf (lapack_int n , double * dl , double * d , double * du , double * du2 , lapack_int * ipiv );
```

```
lapack_int LAPACKE_cgtrf (lapack_int n , lapack_complex_float * dl , lapack_complex_float * d , lapack_complex_float * du , lapack_complex_float * du2 , lapack_int * ipiv );
```

```
lapack_int LAPACKE_zgtrf (lapack_int n , lapack_complex_double * dl , lapack_complex_double * d , lapack_complex_double * du , lapack_complex_double * du2 , lapack_int * ipiv );
```

Include Files

- `mkl.h`

Description

The routine computes the LU factorization of a real or complex tridiagonal matrix A using elimination with partial pivoting and row interchanges.

The factorization has the form

$$A = L*U,$$

where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

Input Parameters

n	The order of the matrix A ; $n \geq 0$.
dl, d, du	Arrays containing elements of A . The array dl of dimension $(n - 1)$ contains the subdiagonal elements of A . The array d of dimension n contains the diagonal elements of A . The array du of dimension $(n - 1)$ contains the superdiagonal elements of A .

Output Parameters

dl	Overwritten by the $(n-1)$ multipliers that define the matrix L from the LU factorization of A .
d	Overwritten by the n diagonal elements of the upper triangular matrix U from the LU factorization of A .
du	Overwritten by the $(n-1)$ elements of the first superdiagonal of U .
$du2$	Array, dimension $(n - 2)$. On exit, $du2$ contains $(n-2)$ elements of the second superdiagonal of U .
$ipiv$	Array, dimension (n) . The pivot indices: for $1 \leq i \leq n$, row i was interchanged with row $ipiv[i-1]$. $ipiv[i-1]$ is always i or $i+1$; $ipiv[i-1] = i$ indicates a row interchange was not required.

Return Values

This function returns a value *info*.

If $info = 0$, the execution is successful.

If $info = -i$, parameter i had an illegal value.

If $info = i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by zero will occur if you use the factor U for solving a system of linear equations.

Application Notes

`?gbtrs`

to solve $A * X = B$ or $A^T * X = B$ or $A^H * X = B$

`?gbcon`

to estimate the condition number of A .

`?dtttrfb`

Computes the factorization of a diagonally dominant tridiagonal matrix.

Syntax

```
void sdttrfb (const MKL_INT * n , float * dl , float * d , const float * du , MKL_INT * info );
```

```
void ddttrfb (const MKL_INT * n , double * dl , double * d , const double * du , MKL_INT * info );
```

```
void cdttrfb (const MKL_INT * n , MKL_Complex8 * dl , MKL_Complex8 * d , const MKL_Complex8 * du , MKL_INT * info );
```

```
void zdttrfb_ (const MKL_INT * n , MKL_Complex16 * dl , MKL_Complex16 * d , const MKL_Complex16 * du , MKL_INT * info );
```

Include Files

- `mk1.h`

Description

The `?dtttrfb` routine computes the factorization of a real or complex tridiagonal matrix A with the BABE (Burning At Both Ends) algorithm without pivoting. The factorization has the form

$$A = L_1 * U * L_2$$

where

- L_1 and L_2 are unit lower bidiagonal with k and $n - k - 1$ subdiagonal elements, respectively, where $k = n/2$, and
- U is an upper bidiagonal matrix with nonzeros in only the main diagonal and first superdiagonal.

Input Parameters

n

The order of the matrix A ; $n \geq 0$.

dl, d, du

Arrays containing elements of A .

The array dl of dimension $(n - 1)$ contains the subdiagonal elements of A .

The array d of dimension n contains the diagonal elements of A .

The array du of dimension $(n - 1)$ contains the superdiagonal elements of A .

Output Parameters

dl

Overwritten by the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A .

<i>d</i>	Overwritten by the <i>n</i> diagonal element reciprocals of the upper triangular matrix <i>U</i> from the factorization of <i>A</i> .
<i>du</i>	Overwritten by the (<i>n</i> -1) elements of the superdiagonal of <i>U</i> .
<i>info</i>	<p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = <i>i</i>, <i>u_{ii}</i> is 0. The factorization has been completed, but <i>U</i> is exactly singular. Division by zero will occur if you use the factor <i>U</i> for solving a system of linear equations.</p>

Application Notes

A diagonally dominant tridiagonal system is defined such that $|d_i| > |dl_{i-1}| + |du_i|$ for any *i*:

$1 < i < n$, and $|d_1| > |du_1|$, $|d_n| > |dl_{n-1}|$

The underlying BABE algorithm is designed for diagonally dominant systems. Such systems are free from the numerical stability issue unlike the canonical systems that use elimination with partial pivoting (see [?gttrf](#)). The diagonally dominant systems are much faster than the canonical systems.

NOTE

- The current implementation of BABE has a potential accuracy issue on very small or large data close to the underflow or overflow threshold respectively. Scale the matrix before applying the solver in the case of such input data.
- Applying the [?dttrfb](#) factorization to non-diagonally dominant systems may lead to an accuracy loss, or false singularity detected due to no pivoting.

?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.

Syntax

```
lapack_int LAPACKE_spotrf (int matrix_layout , char uplo , lapack_int n , float * a ,
lapack_int lda );

lapack_int LAPACKE_dpotrf (int matrix_layout , char uplo , lapack_int n , double * a ,
lapack_int lda );

lapack_int LAPACKE_cpotrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda );

lapack_int LAPACKE_zpotrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda );
```

Include Files

- mkl.h

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix *A*:

$A = U^T * U$ for real data, $A = U^H * U$ for complex data if `uplo='U'`
 $A = L * L^T$ for real data, $A = L * L^H$ for complex data if `uplo='L'`

where L is a lower triangular matrix and U is upper triangular.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <code>uplo = 'U'</code> , the array a stores the upper triangular part of the matrix A , and the strictly lower triangular part of the matrix is not referenced. If <code>uplo = 'L'</code> , the array a stores the lower triangular part of the matrix A , and the strictly upper triangular part of the matrix is not referenced.
<code>n</code>	Specifies the order of the matrix A . The value of n must be at least zero.
<code>a</code>	Array, size $\max(1, lda * n)$. The array a contains either the upper or the lower triangular part of the matrix A (see <code>uplo</code>).
<code>lda</code>	The leading dimension of a . Must be at least $\max(1, n)$.

Output Parameters

<code>a</code>	The upper or lower triangular part of a is overwritten by the Cholesky factor U or L , as specified by <code>uplo</code> .
----------------	--

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, parameter i had an illegal value.

If `info = i`, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A .

Application Notes

If `uplo = 'U'`, the computed factor U is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|, |e_{ij}| \leq c(n)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for `uplo = 'L'`.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

<code>?potrs</code>	to solve $A * X = B$
<code>?pocon</code>	to estimate the condition number of A
<code>?potri</code>	to compute the inverse of A .

See Also

`mkl_progress`

Matrix Storage Schemes

`?potrf2`

Computes Cholesky factorization using a recursive algorithm.

Syntax

```
lapack_int LAPACKE_spotrf2 (int matrix_layout, char uplo, lapack_int n, float * a,
lapack_int lda);

lapack_int LAPACKE_dpotrf2 (int matrix_layout, char uplo, lapack_int n, double * a,
lapack_int lda);

lapack_int LAPACKE_cpotrf2 (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float * a, lapack_int lda);

lapack_int LAPACKE_zpotrf2 (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double * a, lapack_int lda);
```

Include Files

- `mkl.h`

Description

`?potrf2` computes the Cholesky factorization of a real or complex symmetric positive definite matrix A using the recursive algorithm.

The factorization has the form

for real flavors:

$A = U^T * U$, if `uplo = 'U'`, or

$A = L * L^T$, if `uplo = 'L'`,

for complex flavors:

$A = U^H * U$, if `uplo = 'U'`,

or $A = L * L^H$, if `uplo = 'L'`,

where U is an upper triangular matrix and L is lower triangular.

This is the recursive version of the algorithm. It divides the matrix into four submatrices:

$$A = \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix}$$

where $A11$ is $n1$ by $n1$ and $A22$ is $n2$ by $n2$, with $n1 = n/2$ and $n2 = n-n1$.

The subroutine calls itself to factor A_{11} . Update and scale A_{21} or A_{12} , update A_{22} then call itself to factor A_{22} .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	= 'U': Upper triangle of A is stored; = 'L': Lower triangle of A is stored.
<i>n</i>	The order of the matrix A . $n \geq 0$.
<i>a</i>	Array, size ($lda*n$). On entry, the symmetric matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.
<i>lda</i>	The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, the factor U or L from the Cholesky factorization. For real flavors: $A = U^T * U$ or $A = L * L^T$; For complex flavors: $A = U^H * U$ or $A = L * L^H$.
----------	--

Return Values

This function returns a value *info*.

= 0: successful exit

< 0: if *info* = $-i$, the i -th argument had an illegal value

> 0: if *info* = i , the leading minor of order i is not positive definite, and the factorization could not be completed.

?pstrf

Computes the Cholesky factorization with complete pivoting of a real symmetric (complex Hermitian) positive semidefinite matrix.

Syntax

```
lapack_int LAPACKE_spstrf( int matrix_layout, char uplo, lapack_int n, float* a,
lapack_int lda, lapack_int* piv, lapack_int* rank, float tol );
```

```

lapack_int LAPACKE_dpstrf( int matrix_layout, char uplo, lapack_int n, double* a,
lapack_int lda, lapack_int* piv, lapack_int* rank, double tol );

lapack_int LAPACKE_cpstrf( int matrix_layout, char uplo, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_int* piv, lapack_int* rank, float tol );

lapack_int LAPACKE_zpstrf( int matrix_layout, char uplo, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_int* piv, lapack_int* rank, double
tol );

```

Include Files

- mkl.h

Description

The routine computes the Cholesky factorization with complete pivoting of a real symmetric (complex Hermitian) positive semidefinite matrix. The form of the factorization is:

$$\begin{aligned}
 P^T * A * P &= U^T * U, \text{ if } uplo = 'U' \text{ for real flavors,} \\
 P^T * A * P &= U^H * U, \text{ if } uplo = 'U' \text{ for complex flavors,} \\
 P^T * A * P &= L * L^T, \text{ if } uplo = 'L' \text{ for real flavors,} \\
 P^T * A * P &= L * L^H, \text{ if } uplo = 'L' \text{ for complex flavors,}
 \end{aligned}$$

where P is a permutation matrix stored as vector piv , and U and L are upper and lower triangular matrices, respectively.

This algorithm does not attempt to check that A is positive semidefinite. This version of the algorithm calls level 3 BLAS.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the array a stores the upper triangular part of the matrix A , and the strictly lower triangular part of the matrix is not referenced. If $uplo = 'L'$, the array a stores the lower triangular part of the matrix A , and the strictly upper triangular part of the matrix is not referenced.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>a</i>	Array a , size $\max(1, lda * n)$. The array a contains either the upper or the lower triangular part of the matrix A (see <i>uplo</i>).
<i>tol</i>	User defined tolerance. If $tol < 0$, then $n * \epsilon * \max(A_{k,k})$, where ϵ is the machine precision, will be used (see Error Analysis for the definition of machine precision). The algorithm terminates at the $(k-1)$ -st step, if the pivot $\leq tol$.
<i>lda</i>	The leading dimension of a ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	If <i>info</i> = 0, the factor <i>U</i> or <i>L</i> from the Cholesky factorization is as described in <i>Description</i> .
<i>piv</i>	Array, size at least $\max(1, n)$. The array <i>piv</i> is such that the nonzero entries are $P_{piv[k-1], k}$ ($1 \leq k \leq n$).
<i>rank</i>	The rank of <i>a</i> given by the number of steps the algorithm completed.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*k*, the *k*-th argument had an illegal value.

If *info* > 0, the matrix *A* is either rank deficient with a computed rank as returned in *rank*, or is not positive semidefinite.

See Also

Matrix Storage Schemes

?pftrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using the Rectangular Full Packed (RFP) format .

Syntax

```
lapack_int LAPACKE_spftrf (int matrix_layout , char transr , char uplo , lapack_int n ,
float * a );
```

```
lapack_int LAPACKE_dpftrf (int matrix_layout , char transr , char uplo , lapack_int n ,
double * a );
```

```
lapack_int LAPACKE_cpftrf (int matrix_layout , char transr , char uplo , lapack_int n ,
lapack_complex_float * a );
```

```
lapack_int LAPACKE_zpftrf (int matrix_layout , char transr , char uplo , lapack_int n ,
lapack_complex_double * a );
```

Include Files

- mkl.h

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, a Hermitian positive-definite matrix *A*:

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo = 'U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo = 'L' \end{aligned}$$

where *L* is a lower triangular matrix and *U* is upper triangular.

The matrix *A* is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

This is the block version of the algorithm, calling Level 3 BLAS.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>transr</i>	Must be 'N', 'T' (for real data) or 'C' (for complex data). If <i>transr</i> = 'N', the Normal <i>transr</i> of RFP <i>A</i> is stored. If <i>transr</i> = 'T', the Transpose <i>transr</i> of RFP <i>A</i> is stored. If <i>transr</i> = 'C', the Conjugate-Transpose <i>transr</i> of RFP <i>A</i> is stored.
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is stored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix <i>A</i> . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ; $n \geq 0$.
<i>a</i>	Array, size $(n * (n + 1) / 2)$. The array <i>a</i> contains the matrix <i>A</i> in the RFP format.

Output Parameters

<i>a</i>	<i>a</i> is overwritten by the Cholesky factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> and <i>trans</i> .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

See Also

Matrix Storage Schemes

?pptrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using packed storage.

Syntax

```
lapack_int LAPACKE_spptrf (int matrix_layout , char uplo , lapack_int n , float * ap );
lapack_int LAPACKE_dpptrf (int matrix_layout , char uplo , lapack_int n , double *
ap );
lapack_int LAPACKE_cpptrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * ap );
```

```
lapack_int LAPACKE_zpptrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * ap );
```

Include Files

- mkl.h

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite packed matrix A :

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo = 'U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo = 'L' \end{aligned}$$

where L is a lower triangular matrix and U is upper triangular.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is packed in the array <i>ap</i> , and how A is factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the matrix A , and A is factored as $U^H * U$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the matrix A ; A is factored as $L * L^H$.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>ap</i>	Array, size at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains either the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in packed storage (see Matrix Storage Schemes).

Output Parameters

<i>ap</i>	Overwritten by the Cholesky factor U or L , as specified by <i>uplo</i> .
-----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the leading minor of order *i* (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A .

Application Notes

If $uplo = 'U'$, the computed factor U is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|, |e_{ij}| \leq c(n)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for $uplo = 'L'$.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

<code>?pptrs</code>	to solve $A * X = B$
<code>?ppcon</code>	to estimate the condition number of A
<code>?pptri</code>	to compute the inverse of A .

See Also

[mkl_progress](#)

Matrix Storage Schemes

`?pbtrf`

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite band matrix.

Syntax

```
lapack_int LAPACKE_spbtrf (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , float * ab , lapack_int ldab );

lapack_int LAPACKE_dpbtrf (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , double * ab , lapack_int ldab );

lapack_int LAPACKE_cpbtrf (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , lapack_complex_float * ab , lapack_int ldab );

lapack_int LAPACKE_zpbtrf (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , lapack_complex_double * ab , lapack_int ldab );
```

Include Files

- `mkl.h`

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite band matrix A :

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo = 'U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo = 'L' \end{aligned}$$

where L is a lower triangular matrix and U is upper triangular.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is stored in the array <i>ab</i> , and how <i>A</i> is factored: If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	The order of matrix <i>A</i> ; $n \geq 0$.
<i>kd</i>	The number of superdiagonals or subdiagonals in the matrix <i>A</i> ; $kd \geq 0$.
<i>ab</i>	Array, size $\max(1, ldab*n)$. The array <i>ab</i> contains either the upper or the lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i>) in band storage (see Matrix Storage Schemes).
<i>ldab</i>	The leading dimension of the array <i>ab</i> . ($ldab \geq kd + 1$)

Output Parameters

<i>ab</i>	The upper or lower triangular part of <i>A</i> (in band storage) is overwritten by the Cholesky factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> .
-----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix *A*.

Application Notes

If *uplo* = 'U', the computed factor *U* is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(kd + 1)\varepsilon |U^H| |U|, |e_{ij}| \leq c(kd + 1)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for *uplo* = 'L'.

The total number of floating-point operations for real flavors is approximately $n(kd+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that kd is much less than n .

After calling this routine, you can call the following routines:

[?pbtrs](#) to solve $A*X = B$

`?pbcon`to estimate the condition number of A .

See Also

`mkl_progress`

Matrix Storage Schemes

`?pttrf`*Computes the factorization of a symmetric (Hermitian) positive-definite tridiagonal matrix.*

Syntax

```
lapack_int LAPACKE_spttrf( lapack_int n, float* d, float* e );
lapack_int LAPACKE_dpttrf( lapack_int n, double* d, double* e );
lapack_int LAPACKE_cpttrf( lapack_int n, float* d, lapack_complex_float* e );
lapack_int LAPACKE_zpttrf( lapack_int n, double* d, lapack_complex_double* e );
```

Include Files

- `mkl.h`

Description

The routine forms the factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite tridiagonal matrix A :

$A = L * D * L^T$ for real flavors, or

$A = L * D * L^H$ for complex flavors,

where D is diagonal and L is unit lower bidiagonal. The factorization may also be regarded as having the form $A = U^T * D * U$ for real flavors, or $A = U^H * D * U$ for complex flavors, where U is unit upper bidiagonal.

Input Parameters

n	The order of the matrix A ; $n \geq 0$.
d	Array, dimension (n) . Contains the diagonal elements of A .
e	Array, dimension $(n - 1)$. Contains the subdiagonal elements of A .

Output Parameters

d	Overwritten by the n diagonal elements of the diagonal matrix D from the $L * D * L^T$ (for real flavors) or $L * D * L^H$ (for complex flavors) factorization of A .
e	Overwritten by the $(n - 1)$ sub-diagonal elements of the unit bidiagonal factor L or U from the factorization of A .

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If $info = -i$, parameter i had an illegal value.

If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite; if $i < n$, the factorization could not be completed, while if $i = n$, the factorization was completed, but $d[n - 1] \leq 0$.

?sytrf

Computes the Bunch-Kaufman factorization of a symmetric matrix.

Syntax

```
lapack_int LAPACKESsytrf (int matrix_layout , char uplo , lapack_int n , float * a ,
lapack_int lda , lapack_int * ipiv );

lapack_int LAPACKESdytrf (int matrix_layout , char uplo , lapack_int n , double * a ,
lapack_int lda , lapack_int * ipiv );

lapack_int LAPACKESsytrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , lapack_int * ipiv );

lapack_int LAPACKESzsytrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , lapack_int * ipiv );
```

Include Files

- mkl.h

Description

The routine computes the factorization of a real/complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

if $uplo = 'U'$, $A = U * D * U^T$
 if $uplo = 'L'$, $A = L * D * L^T$

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE This routine supports the Progress Routine feature. See [Progress Routine](#) for details.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If $uplo = 'U'$, the array a stores the upper triangular part of the matrix A , and A is factored as $U * D * U^T$.

If `uplo = 'L'`, the array `a` stores the lower triangular part of the matrix `A`, and `A` is factored as $L^*D^*L^T$.

`n`

The order of matrix `A`; $n \geq 0$.

`a`

Array, size $\max(1, lda * n)$. The array `a` contains either the upper or the lower triangular part of the matrix `A` (see `uplo`).

`lda`

The leading dimension of `a`; at least $\max(1, n)$.

Output Parameters

`a`

The upper or lower triangular part of `a` is overwritten by details of the block-diagonal matrix `D` and the multipliers used to obtain the factor `U` (or `L`).

`ipiv`

Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of `D`. If `ipiv[i-1] = k > 0`, then d_{ii} is a 1-by-1 block, and the i -th row and column of `A` was interchanged with the k -th row and column.

If `uplo = 'U'` and `ipiv[i] = ipiv[i-1] = -m < 0`, then `D` has a 2-by-2 block in rows/columns i and $i+1$, and i -th row and column of `A` was interchanged with the m -th row and column.

If `uplo = 'L'` and `ipiv[i] = ipiv[i-1] = -m < 0`, then `D` has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of `A` was interchanged with the m -th row and column.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, parameter i had an illegal value.

If `info = i`, D_{ii} is 0. The factorization has been completed, but `D` is exactly singular. Division by 0 will occur if you use `D` for solving a system of linear equations.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of `U` and `L` are not stored. The remaining elements of `U` and `L` are stored in the corresponding columns of the array `a`, but additional row interchanges are required to recover `U` or `L` explicitly (which is seldom necessary).

If `ipiv[i-1] = i` for all $i = 1 \dots n$, then all off-diagonal elements of `U` (`L`) are stored explicitly in the corresponding elements of the array `a`.

If `uplo = 'U'`, the computed factors `U` and `D` are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. A similar estimate holds for the computed `L` and `D` when `uplo = 'L'`.

The total number of floating-point operations is approximately $(1/3) n^3$ for real flavors or $(4/3) n^3$ for complex flavors.

After calling this routine, you can call the following routines:

`?sytrs` to solve $A * X = B$
`?sycon` to estimate the condition number of A
`?sytri` to compute the inverse of A .

If `uplo = 'U'`, then $A = U * D * U'$, where

$U = P(n) * U(n) * \dots * P(k) * U(k) * \dots,$

that is, U is a product of terms $P(k) * U(k)$, where

- k decreases from n to 1 in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by `ipiv[k-1]`.
- $U(k)$ is a unit upper triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$U(k) = \begin{pmatrix} I & v & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \begin{matrix} k - s \\ s \\ n - k \end{matrix}$$

$k - s \quad s \quad n - k$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(1:k-1,k)$.

If $s = 2$, the upper triangle of $D(k)$ overwrites $A(k-1,k-1)$, $A(k-1,k)$ and $A(k,k)$, and v overwrites $A(1:k-2,k-1:k)$.

If `uplo = 'L'`, then $A = L * D * L'$, where

$L = P(1) * L(1) * \dots * P(k) * L(k) * \dots,$

that is, L is a product of terms $P(k) * L(k)$, where

- k increases from 1 to n in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by `ipiv(k)`.
- $L(k)$ is a unit lower triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$L(k) = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & v & I \end{pmatrix} \begin{matrix} k-1 \\ s \\ n-k-s+1 \end{matrix}$$

$k-1 \quad s \quad n-k-s+1$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $D(k)$ overwrites $A(k,k)$, $A(k+1,k)$, and $A(k+1,k+1)$, and v overwrites $A(k+2:n,k:k+1)$.

See Also

[mkl_progress](#)

Matrix Storage Schemes

?sytrf_aa

Computes the factorization of a symmetric matrix using Aasen's algorithm.

```
lapack_int LAPACK_essytrf_aa (int matrix_layout, char uplo, lapack_int n, float * A,
lapack_int lda, lapack_int * ipiv);

lapack_int LAPACK_dsytrf_aa (int matrix_layout, char uplo, lapack_int n, double * A,
lapack_int lda, lapack_int * ipiv);

lapack_int LAPACK_csytrf_aa (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float * A, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACK_zsytrf_aa (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double * A, lapack_int lda, lapack_int * ipiv);
```

Description

?sytrf_aa computes the factorization of a symmetric matrix A using Aasen's algorithm. The form of the factorization is $A = U^*T^*U^T$ or $A = L^*T^*L^T$ where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and T is a complex symmetric tridiagonal matrix.

This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	<ul style="list-style-type: none"> = 'U': The upper triangle of A is stored. = 'L': The lower triangle of A is stored.
<i>n</i>	The order of the matrix A . $n \geq 0$.
<i>A</i>	Array of size $\max(1, lda*n)$. The array A contains either the upper or the lower triangular part of the matrix A (see <i>uplo</i>).
<i>lda</i>	The leading dimension of the array A .

Output Parameters

A	On exit, the tridiagonal matrix is stored in the diagonals and the subdiagonals of A just below (or above) the diagonals, and L is stored below (or above) the subdiagonals, when $uplo$ is 'L' (or 'U').
$ipiv$	Array of size n . On exit, it contains the details of the interchanges; that is, the row and column k of A were interchanged with the row and column $ipiv(k)$.

Return Values

This function returns a value $info$.

= 0: Successful exit.

< 0: If $info = -i$, the i^{th} argument had an illegal value.

> 0: If $info = i$, $D(i,i)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?sytrf_rook

Computes the bounded Bunch-Kaufman factorization of a symmetric matrix.

Syntax

```
lapack_int LAPACKE_ssytrf_rook (int matrix_layout, char uplo, lapack_int n, float * a,
lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_dsytrf_rook (int matrix_layout, char uplo, lapack_int n, double * a,
lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_csytrf_rook (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float * a, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_zsytrf_rook (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double * a, lapack_int lda, lapack_int * ipiv);
```

Include Files

- mkl.h

Description

The routine computes the factorization of a real/complex symmetric matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The form of the factorization is:

if $uplo = 'U'$, $A = U^* D^* U^T$
 if $uplo = 'L'$, $A = L^* D^* L^T$,

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout for array <i>b</i> is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is stored and how <i>A</i> is factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix <i>A</i> , and <i>A</i> is factored as $U^*D^*U^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix <i>A</i> , and <i>A</i> is factored as $L^*D^*L^T$.
<i>n</i>	The order of matrix <i>A</i> ; $n \geq 0$.
<i>a</i>	Array, size <i>lda</i> * <i>n</i> . The array <i>a</i> contains either the upper or the lower triangular part of the matrix <i>A</i> (see <i>uplo</i>).
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by details of the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>).
<i>ipiv</i>	If <i>ipiv</i> (<i>k</i>) > 0, then rows and columns <i>k</i> and <i>ipiv</i> (<i>k</i>) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block. If <i>uplo</i> = 'U' and <i>ipiv</i> (<i>k</i>) < 0 and <i>ipiv</i> (<i>k</i> - 1) < 0, then rows and columns <i>k</i> and - <i>ipiv</i> (<i>k</i>) were interchanged, rows and columns <i>k</i> - 1 and - <i>ipiv</i> (<i>k</i> - 1) were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block. If <i>uplo</i> = 'L' and <i>ipiv</i> (<i>k</i>) < 0 and <i>ipiv</i> (<i>k</i> + 1) < 0, then rows and columns <i>k</i> and - <i>ipiv</i> (<i>k</i>) were interchanged, rows and columns <i>k</i> + 1 and - <i>ipiv</i> (<i>k</i> + 1) were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, D_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

Application Notes

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

<code>?sytrs_rook</code>	to solve $A^*X = B$
--------------------------	---------------------

?sycon_rook (Fortran only) to estimate the condition number of A
 ?sytri_rook (Fortran only) to compute the inverse of A .

If $uplo = 'U'$, then $A = U^*D^*U$, where

$$U = P(n) * U(n) * \dots * P(k) * U(k) * \dots,$$

that is, U is a product of terms $P(k) * U(k)$, where

- k decreases from n to 1 in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by $ipiv[k-1]$.
- $U(k)$ is a unit upper triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$U(k) = \begin{pmatrix} I & v & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \begin{matrix} k-s \\ s \\ n-k \end{matrix}$$

$k-s \quad s \quad n-k$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(1:k-1,k)$.

If $s = 2$, the upper triangle of $D(k)$ overwrites $A(k-1,k-1)$, $A(k-1,k)$ and $A(k,k)$, and v overwrites $A(1:k-2,k-1:k)$.

If $uplo = 'L'$, then $A = L^*D^*L$, where

$$L = P(1) * L(1) * \dots * P(k) * L(k) * \dots,$$

that is, L is a product of terms $P(k) * L(k)$, where

- k increases from 1 to n in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by $ipiv(k)$.
- $L(k)$ is a unit lower triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$L(k) = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & v & I \end{pmatrix} \begin{matrix} k-1 \\ s \\ n-k-s+1 \end{matrix}$$

$k-1 \quad s \quad n-k-s+1$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $D(k)$ overwrites $A(k,k)$, $A(k+1,k)$, and $A(k+1,k+1)$, and v overwrites $A(k+2:n,k:k+1)$.

See Also

Matrix Storage Schemes

`?sytrf_rk`

Computes the factorization of a real or complex symmetric indefinite matrix using the bounded Bunch-Kaufman (rook) diagonal pivoting method (BLAS3 blocked algorithm).

```
lapack_int LAPACK_essytrf_rk (int matrix_layout, char uplo, lapack_int n, float * A,
lapack_int lda, float * e, lapack_int * ipiv);

lapack_int LAPACK_dsytrf_rk (int matrix_layout, char uplo, lapack_int n, double * A,
lapack_int lda, double * e, lapack_int * ipiv);

lapack_int LAPACK_csytrf_rk (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float * A, lapack_int lda, lapack_complex_float * e, lapack_int * ipiv);

lapack_int LAPACK_zsytrf_rk (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double * A, lapack_int lda, lapack_complex_double * e, lapack_int *
ipiv);
```

Description

`?sytrf_rk` computes the factorization of a real or complex symmetric matrix A using the bounded Bunch-Kaufman (rook) diagonal pivoting method: $A = P*U*D*(U^T)*(P^T)$ or $A = P*L*D*(L^T)*(P^T)$, where U (or L) is unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of U (or L), P is a permutation matrix, P^T is the transpose of P , and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level-3 BLAS.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Specifies whether the upper or lower triangular part of the symmetric matrix A is stored: <ul style="list-style-type: none"> <code>'U'</code>: Upper triangular <code>'L'</code>: Lower triangular
<code>n</code>	The order of the matrix A . $n \geq 0$.
<code>A</code>	Array of size $\max(1, lda*n)$. On entry, the symmetric matrix A . If <code>uplo = 'U'</code> , the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced. If <code>uplo = 'L'</code> , the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.
<code>lda</code>	The leading dimension of the array A .

Output Parameters

<code>A</code>	On exit, contains:
----------------	--------------------

- Only diagonal elements of the symmetric block diagonal matrix D on the diagonal of A; that is, $D(k,k) = A(k,k)$; (superdiagonal (or subdiagonal) elements of D are stored on exit in array *e*).
- If *uplo* = 'U', factor U in the superdiagonal part of A. If *uplo* = 'L', factor L in the subdiagonal part of A.

e

Array of size *n*. On exit, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = 'U', $e(i) = D(i-1,i)$, $i=2:N$, and $e(1)$ is set to 0. If *uplo* = 'L', $e(i) = D(i+1,i)$, $i=1:N-1$, and $e(n)$ is set to 0.

NOTE For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e[k-1]$ is set to 0 in both the *uplo* = 'U' and *uplo* = 'L' cases.

ipiv

Array of size *n*. *ipiv* describes the permutation matrix P in the factorization of matrix A as follows: The absolute value of *ipiv*(*k*) represents the index of the row and column that were interchanged with the *k*th row and column. The value of *uplo* describes the order in which the interchanges were applied. Also, the sign of *ipiv* represents the block structure of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks, which correspond to 1 or 2 interchanges at each factorization step. If *uplo* = 'U' (in factorization order, *k* decreases from *n* to 1):

1. A single positive entry *ipiv*(*k*) > 0 means that $D(k,k)$ is a 1-by-1 diagonal block. If *ipiv*(*k*) != *k*, rows and columns *k* and *ipiv*(*k*) were interchanged in the matrix $A(1:N,1:N)$. If *ipiv*(*k*) = *k*, no interchange occurred.
2. A pair of consecutive negative entries *ipiv*(*k*) < 0 and *ipiv*(*k*-1) < 0 means that $D(k-1:k,k-1:k)$ is a 2-by-2 diagonal block. (Note that negative entries in *ipiv* appear *only* in pairs.)
 - If -*ipiv*(*k*) != *k*, rows and columns *k* and -*ipiv*(*k*) were interchanged in the matrix $A(1:N,1:N)$. If -*ipiv*(*k*) = *k*, no interchange occurred.
 - If -*ipiv*(*k*-1) != *k*-1, rows and columns *k*-1 and -*ipiv*(*k*-1) were interchanged in the matrix $A(1:N,1:N)$. If -*ipiv*(*k*-1) = *k*-1, no interchange occurred.
3. In both cases 1 and 2, always $ABS(ipiv(k)) \leq k$.

NOTE Any entry *ipiv*(*k*) is always nonzero on output.

If *uplo* = 'L' (in factorization order, *k* increases from 1 to *n*):

1. A single positive entry *ipiv*(*k*) > 0 means that $D(k,k)$ is a 1-by-1 diagonal block. If *ipiv*(*k*) != *k*, rows and columns *k* and *ipiv*(*k*) were interchanged in the matrix $A(1:N,1:N)$. If *ipiv*(*k*) = *k*, no interchange occurred.
2. A pair of consecutive negative entries *ipiv*(*k*) < 0 and *ipiv*(*k*+1) < 0 means that $D(k:k+1,k:k+1)$ is a 2-by-2 diagonal block. (Note that negative entries in *ipiv* appear *only* in pairs.)

- If $-ipiv(k) \neq k$, rows and columns k and $-ipiv(k)$ were interchanged in the matrix $A(1:N,1:N)$. If $-ipiv(k) = k$, no interchange occurred.
 - If $-ipiv(k+1) \neq k+1$, rows and columns $k-1$ and $-ipiv(k-1)$ were interchanged in the matrix $A(1:N,1:N)$. If $-ipiv(k+1) = k+1$, no interchange occurred.
3. In both cases 1 and 2, always $ABS(ipiv(k)) \geq k$.

NOTE Any entry $ipiv(k)$ is always nonzero on output.

Return Values

This function returns a value *info*.

= 0: Successful exit.

< 0: If *info* = $-k$, the k^{th} argument had an illegal value.

> 0: If *info* = k , the matrix A is singular. If *uplo* = 'U', column k in the upper triangular part of A contains all zeros. If *uplo* = 'L', column k in the lower triangular part of A contains all zeros. Therefore, $D(k,k)$ is exactly zero, and superdiagonal elements of column k of U (or subdiagonal elements of column k of L) are all zeros. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

zhetrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax

```
lapack_int LAPACKC_chetrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , lapack_int * ipiv );

lapack_int LAPACKC_zhetrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , lapack_int * ipiv );
```

Include Files

- mkl.h

Description

The routine computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method:

```
if uplo='U',  $A = U^H D U$ 
if uplo='L',  $A = L^H D L$ ,
```

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Routine](#) for details.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A , and A is factored as $U^*D^*U^H$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A , and A is factored as $L^*D^*L^H$.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>a</i>	Array, size $\max(1, lda * n)$. The array <i>a</i> contains the upper or the lower triangular part of the matrix A (see <i>uplo</i>).
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).
<i>ipiv</i>	Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D . If <i>ipiv</i> [<i>i</i> -1] = <i>k</i> > 0, then d_{ii} is a 1-by-1 block, and the <i>i</i> -th row and column of A was interchanged with the <i>k</i> -th row and column. If <i>uplo</i> = 'U' and <i>ipiv</i> [<i>i</i>] = <i>ipiv</i> [<i>i</i> -1] = - <i>m</i> < 0, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and <i>i</i> -th row and column of A was interchanged with the <i>m</i> -th row and column. If <i>uplo</i> = 'L' and <i>ipiv</i> [<i>i</i>] = <i>ipiv</i> [<i>i</i> -1] = - <i>m</i> < 0, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and (<i>i</i> +1)-th row and column of A was interchanged with the <i>m</i> -th row and column.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

Application Notes

This routine is suitable for Hermitian matrices that are not known to be positive-definite. If A is in fact positive-definite, the routine does not perform interchanges, and no 2-by-2 diagonal blocks occur in D .

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array a , but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv[i-1] = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array a .

If $uplo = 'U'$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for the computed L and D when $uplo = 'L'$.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

<code>?hetrs</code>	to solve $A * X = B$
<code>?hecon</code>	to estimate the condition number of A
<code>?hetri</code>	to compute the inverse of A .

See Also

[mkl_progress](#)

Matrix Storage Schemes

`?hetrf_aa`

Computes the factorization of a complex hermitian matrix using Aasen's algorithm.

```
LAPACK_DECL lapack_int LAPACKE_chetrf_aa (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float * a, lapack_int lda, lapack_int * ipiv );
```

```
LAPACK_DECL lapack_int LAPACKE_zhetrf_aa (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double * a, lapack_int lda, lapack_int * ipiv );
```

Description

`?hetrf_aa` computes the factorization of a complex Hermitian matrix A using Aasen's algorithm. The form of the factorization is $A = U * T * U^H$ or $a = L * T * L^H$ where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and T is a Hermitian tridiagonal matrix. This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	= 'U': Upper triangle of A is stored; = 'L': Lower triangle of a is stored.
<code>n</code>	The order of the matrix A . $n \geq 0$.
<code>a</code>	Array of size <code>lda*n</code> . On entry, the Hermitian matrix A .

If `uplo = 'U'`, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced.

If `uplo = 'L'`, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

`lda`

The leading dimension of the array a . $lda \geq \max(1, n)$.

`lwork`

See [Syntax - Workspace](#). The length of `work`. $lwork \geq 2 * n$. For optimum performance $lwork \geq n * (1 + nb)$, where nb is the optimal block size. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by xerbla.

Output Parameters

`a`

On exit, the tridiagonal matrix is stored in the diagonals and the subdiagonals of a just below (or above) the diagonals, and L is stored below (or above) the subdiagonals, when `uplo` is 'L' (or 'U').

`ipiv`

array, dimension (n) On exit, it contains the details of the interchanges: the row and column k of a were interchanged with the row and column `ipiv[k]`.

`work`

See [Syntax - Workspace](#). Array of size $(\max(1, lwork))$. On exit, if `info = 0`, `work[0]` returns the optimal `lwork`.

Return Values

This function returns a value `info`.

If `info = 0`: successful exit < 0: if `info = -i`, the i -th argument had an illegal value,

If `info > 0`: if `info = i`, $D_{i,i}$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

Syntax - Workspace

Use this interface if you want to explicitly provide the workspace array.

```
LAPACK_DECL lapack_int LAPACKE_chetrf_aa_work (int matrix_layout, char uplo, lapack_int
n, lapack_complex_float * a, lapack_int lda, lapack_int * ipiv, lapack_complex_float *
work, lapack_int lwork );
```

```
LAPACK_DECL lapack_int LAPACKE_zhetrf_aa_work (int matrix_layout, char uplo, lapack_int
n, lapack_complex_double * a, lapack_int lda, lapack_int * ipiv, lapack_complex_double
* work, lapack_int lwork );
```

?hetrf_rook

Computes the bounded Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax

```
lapack_int LAPACKE_chetrf_rook (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float * a, lapack_int lda, lapack_int * ipiv);
```

```
lapack_int LAPACKE_zhetrf_rook (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double * a, lapack_int lda, lapack_int * ipiv);
```

Include Files

- mkl.h

Description

The routine computes the factorization of a complex Hermitian matrix A using the bounded Bunch-Kaufman diagonal pivoting method:

```
if uplo='U',  $A = U^H D U$ 
if uplo='L',  $A = L D L^H$ ,
```

where A is the input matrix, U (or L) is a product of permutation and unit upper (or lower) triangular matrices, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout for array <i>a</i> is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A .
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>a</i>	Array <i>a</i> , size (<i>lda</i> * <i>n</i>) The array <i>a</i> contains the upper or the lower triangular part of the matrix A (see <i>uplo</i>). If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix A , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix A , and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	The block diagonal matrix D and the multipliers used to obtain the factor U or L (see Application Notes for further details).
<i>ipiv</i>	<ul style="list-style-type: none"> • If <i>uplo</i> = 'U': If <i>ipiv</i>(<i>k</i>) > 0, then rows and columns <i>k</i> and <i>ipiv</i>(<i>k</i>) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged and rows and columns $k - 1$ and $-ipiv(k - 1)$ were interchanged, $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.

- If $uplo = 'L'$:

If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged and rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

Return Values

This function returns a value *info*.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, D_{ii} is exactly 0. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by 0 will occur if you use D for solving a system of linear equations.

Application Notes

If $uplo = 'U'$, then $A = U^*D^*U^H$, where

$U = P(n)*U(n)* \dots *P(k)U(k)* \dots$,

i.e., U is a product of terms $P(k)*U(k)$, where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$. $P(k)$ is a permutation matrix as defined by $ipiv(k)$, and $U(k)$ is a unit upper triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$U(k) = \begin{matrix} & k-s & s & n-k \\ k-s & \begin{pmatrix} I & v & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \\ s & \\ n-k & \end{matrix}$$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(1:k-1,k)$.

If $s = 2$, the upper triangle of $D(k)$ overwrites $A(k-1,k-1)$, $A(k-1,k)$, and $A(k,k)$, and v overwrites $A(1:k-2,k-1:k)$.

If $uplo = 'L'$, then $A = L^*D^*L^H$, where

$L = P(1)*L(1)* \dots *P(k)*L(k)* \dots$,

i.e., L is a product of terms $P(k)*L(k)$, where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$. $P(k)$ is a permutation matrix as defined by $ipiv(k)$, and $L(k)$ is a unit lower triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$L(k) = \begin{matrix} & k-1 & s & n-k-s+1 \\ k-1 & \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & v & I \end{pmatrix} \\ s & \\ n-k-s+1 & \end{matrix}$$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $D(k)$ overwrites $A(k,k)$, $A(k+1,k)$, and $A(k+1,k+1)$, and v overwrites $A(k+2:n,k:k+1)$.

See Also

[mkl_progress](#)

Matrix Storage Schemes

?hetrf_rk

Computes the factorization of a complex Hermitian indefinite matrix using the bounded Bunch-Kaufman (rook) diagonal pivoting method (BLAS3 blocked algorithm).

```
lapack_int LAPACKE_chetrf_rk (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float * A, lapack_int lda, lapack_complex_float * e, lapack_int * ipiv);

lapack_int LAPACKE_zhetrf_rk (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double * A, lapack_int lda, lapack_complex_double * e, lapack_int *
ipiv);
```

Description

?hetrf_rk computes the factorization of a complex Hermitian matrix A using the bounded Bunch-Kaufman (rook) diagonal pivoting method: $A = P*U*D*(U^H)*(P^T)$ or $A = P*L*D*(L^H)*(P^T)$, where U (or L) is unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of U (or L), P is a permutation matrix, P^T is the transpose of P , and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: <ul style="list-style-type: none"> = 'U': Upper triangular. = 'L': Lower triangular.
<i>n</i>	The order of the matrix A . $n \geq 0$.
<i>A</i>	Array of size $\max(1, lda*n)$. On entry, the Hermitian matrix A . If <i>uplo</i> = 'U': The leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced. If <i>uplo</i> = 'L': The leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.
<i>lda</i>	The leading dimension of the array A .

Output Parameters

<i>A</i>	On exit, contains: <ul style="list-style-type: none"> Only diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A; that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of D are stored on exit in array <i>e</i>.
----------	---

—and—

- If $uplo = 'U'$, factor U in the superdiagonal part of A. If $uplo = 'L'$, factor L in the subdiagonal part of A.

e

Array of size n . On exit, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If $uplo = 'U'$, $e(i) = D(i-1,i)$, $i=2:N$, and $e(1)$ is set to 0. If $uplo = 'L'$, $e(i) = D(i+1,i)$, $i=1:N-1$, and $e(n)$ is set to 0.

NOTE For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e[k-1]$ is set to 0 in both the $uplo = 'U'$ and $uplo = 'L'$ cases.

ipiv

Array of size n . $ipiv$ describes the permutation matrix P in the factorization of matrix A as follows: The absolute value of $ipiv[k-1]$ represents the index of row and column that were interchanged with the k^{th} row and column. The value of $uplo$ describes the order in which the interchanges were applied. Also, the sign of $ipiv$ represents the block structure of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks that correspond to 1 or 2 interchanges at each factorization step. If $uplo = 'U'$ (in factorization order, k decreases from n to 1):

1. A single positive entry $ipiv(k) > 0$ means that $D(k,k)$ is a 1-by-1 diagonal block. If $ipiv(k) \neq k$, rows and columns k and $ipiv(k)$ were interchanged in the matrix $A(1:N,1:N)$. If $ipiv(k) = k$, no interchange occurred.
2. A pair of consecutive negative entries $ipiv(k) < 0$ and $ipiv(k-1) < 0$ means that $D(k-1:k,k-1:k)$ is a 2-by-2 diagonal block. (Note that negative entries in $ipiv$ appear *only* in pairs.)
 - If $-ipiv(k) \neq k$, rows and columns k and $-ipiv(k)$ were interchanged in the matrix $A(1:N,1:N)$. If $-ipiv(k) = k$, no interchange occurred.
 - If $-ipiv(k-1) \neq k-1$, rows and columns $k-1$ and $-ipiv(k-1)$ were interchanged in the matrix $A(1:N,1:N)$. If $-ipiv(k-1) = k-1$, no interchange occurred.
3. In both cases 1 and 2, always $ABS(ipiv(k)) \leq k$.

NOTE Any entry $ipiv(k)$ is always nonzero on output.

If $uplo = 'L'$ (in factorization order, k increases from 1 to n):

1. A single positive entry $ipiv(k) > 0$ means that $D(k,k)$ is a 1-by-1 diagonal block. If $ipiv(k) \neq k$, rows and columns k and $ipiv(k)$ were interchanged in the matrix $A(1:N,1:N)$. If $ipiv(k) = k$, no interchange occurred.
2. A pair of consecutive negative entries $ipiv(k) < 0$ and $ipiv(k+1) < 0$ means that $D(k:k+1,k:k+1)$ is a 2-by-2 diagonal block. (Note that negative entries in $ipiv$ appear *only* in pairs.)

- If $-ipiv(k) \neq k$, rows and columns k and $-ipiv(k)$ were interchanged in the matrix $A(1:N,1:N)$. If $-ipiv(k) = k$, no interchange occurred.
 - If $-ipiv(k+1) \neq k+1$, rows and columns $k-1$ and $-ipiv(k-1)$ were interchanged in the matrix $A(1:N,1:N)$. If $-ipiv(k+1) = k+1$, no interchange occurred.
3. In both cases 1 and 2, always $ABS(ipiv(k)) \geq k$.

NOTE Any entry $ipiv(k)$ is always nonzero on output.

Return Values

This function returns a value *info*.

= 0: Successful exit.

< 0: If *info* = $-k$, the k^{th} argument had an illegal value.

> 0: If *info* = k , the matrix A is singular. If *uplo* = 'U', the column k in the upper triangular part of A contains all zeros. If *uplo* = 'L', the column k in the lower triangular part of A contains all zeros. Therefore $D(k,k)$ is exactly zero, and superdiagonal elements of column k of U (or subdiagonal elements of column k of L) are all zeros. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?sptf

Computes the Bunch-Kaufman factorization of a symmetric matrix using packed storage.

Syntax

```
lapack_int LAPACK_essptf (int matrix_layout , char uplo , lapack_int n , float * ap ,
lapack_int * ipiv );

lapack_int LAPACK_dsptrf (int matrix_layout , char uplo , lapack_int n , double * ap ,
lapack_int * ipiv );

lapack_int LAPACK_csptrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * ap , lapack_int * ipiv );

lapack_int LAPACK_zsptrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * ap , lapack_int * ipiv );
```

Include Files

- mkl.h

Description

The routine computes the factorization of a real/complex symmetric matrix A stored in the packed format using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

```
if uplo='U',  $A = U^T D U$ 
if uplo='L',  $A = L D L^T$ ,
```

where U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is packed in the array <i>ap</i> and how A is factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the matrix A , and A is factored as $U^*D^*U^T$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the matrix A , and A is factored as $L^*D^*L^T$.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>ap</i>	Array, size at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in <i>packed storage</i> (see Matrix Storage Schemes).

Output Parameters

<i>ap</i>	The upper or lower triangle of A (as specified by <i>uplo</i>) is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).
<i>ipiv</i>	Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D . If <i>ipiv</i> [<i>i</i> -1] = $k > 0$, then d_{ii} is a 1-by-1 block, and the <i>i</i> -th row and column of A was interchanged with the <i>k</i> -th row and column. If <i>uplo</i> = 'U' and <i>ipiv</i> [<i>i</i>] = <i>ipiv</i> [<i>i</i> -1] = $-m < 0$, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and <i>i</i> -th row and column of A was interchanged with the <i>m</i> -th row and column. If <i>uplo</i> = 'L' and <i>ipiv</i> [<i>i</i>] = <i>ipiv</i> [<i>i</i> -1] = $-m < 0$, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and (<i>i</i> +1)-th row and column of A was interchanged with the <i>m</i> -th row and column.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L overwrite elements of the corresponding columns of the array ap , but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in packed form.

If $uplo = 'U'$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. A similar estimate holds for the computed L and D when $uplo = 'L'$.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

<code>?spttrs</code>	to solve $A * X = B$
<code>?spcon</code>	to estimate the condition number of A
<code>?sptri</code>	to compute the inverse of A .

See Also

[mkl_progress](#)

Matrix Storage Schemes

`?hptrf`

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix using packed storage.

Syntax

```
lapack_int LAPACKE_chptrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * ap , lapack_int * ipiv );
```

```
lapack_int LAPACKE_zhptrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * ap , lapack_int * ipiv );
```

Include Files

- `mkl.h`

Description

The routine computes the factorization of a complex Hermitian packed matrix A using the Bunch-Kaufman diagonal pivoting method:

```
if uplo='U',  $A = U * D * U^H$ 
if uplo='L',  $A = L * D * L^H$ ,
```

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is packed and how A is factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the matrix A , and A is factored as $U^*D^*U^H$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the matrix A , and A is factored as $L^*D^*L^H$.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>ap</i>	Array, size at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in <i>packed storage</i> (see Matrix Storage Schemes).

Output Parameters

<i>ap</i>	The upper or lower triangle of A (as specified by <i>uplo</i>) is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).
<i>ipiv</i>	Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D . If <i>ipiv</i> [<i>i</i> -1] = $k > 0$, then d_{ii} is a 1-by-1 block, and the <i>i</i> -th row and column of A was interchanged with the <i>k</i> -th row and column. If <i>uplo</i> = 'U' and <i>ipiv</i> [<i>i</i>] = <i>ipiv</i> [<i>i</i> -1] = $-m < 0$, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and <i>i</i> -th row and column of A was interchanged with the <i>m</i> -th row and column. If <i>uplo</i> = 'L' and <i>ipiv</i> [<i>i</i>] = <i>ipiv</i> [<i>i</i> -1] = $-m < 0$, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and (<i>i</i> +1)-th row and column of A was interchanged with the <i>m</i> -th row and column.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the array ap , but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv[i-1] = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array a .

If $uplo = 'U'$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n) \varepsilon P \|U\| \|D\| \|U^T\| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for the computed L and D when $uplo = 'L'$.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

<code>?hptrs</code>	to solve $A * X = B$
<code>?hpcon</code>	to estimate the condition number of A
<code>?hptri</code>	to compute the inverse of A .

See Also

[mkl_progress](#)

Matrix Storage Schemes

`mkl_?spffrt2`, `mkl_?spffrtx`

Computes the partial LDL^T factorization of a symmetric matrix using packed storage.

Syntax

```
void mkl_sspffrt2 (float *ap , const MKL_INT *n , const MKL_INT *ncolm , float *work ,
float *work2 );

void mkl_dspffrt2 (double *ap , const MKL_INT *n , const MKL_INT *ncolm , double
*work , double *work2 );

void mkl_cspffrt2 (MKL_Complex8 *ap , const MKL_INT *n , const MKL_INT *ncolm ,
MKL_Complex8 *work , MKL_Complex8 *work2 );

void mkl_zspffrt2 (MKL_Complex16 *ap , const MKL_INT *n , const MKL_INT *ncolm ,
MKL_Complex16 *work , MKL_Complex16 *work2 );

void mkl_sspffrtx (float *ap , const MKL_INT *n , const MKL_INT *ncolm , float *work ,
float *work2 );

void mkl_dspffrtx (double *ap , const MKL_INT *n , const MKL_INT *ncolm , double
*work , double *work2 );

void mkl_cspffrtx (MKL_Complex8 *ap , const MKL_INT *n , const MKL_INT *ncolm ,
MKL_Complex8 *work , MKL_Complex8 *work2 );

void mkl_zspffrtx (MKL_Complex16 *ap , const MKL_INT *n , const MKL_INT *ncolm ,
MKL_Complex16 *work , MKL_Complex16 *work2 );
```

Include Files

- `mk1.h`

Description

The routine computes the partial factorization $A = LDL^T$, where L is a lower triangular matrix and D is a diagonal matrix.

Caution

The routine assumes that the matrix A is factorizable. The routine does not perform pivoting and does not handle diagonal elements which are zero, which cause the routine to produce incorrect results without any indication.

Consider the matrix $A = \begin{pmatrix} a & b^T \\ b & C \end{pmatrix}$, where a is the element in the first row and first column of A , b is a column vector of size $n - 1$ containing the elements from the second through n -th column of A , C is the lower-right square submatrix of A , and I is the identity matrix.

The `mk1_?spffrt2` routine performs $ncolm$ successive factorizations of the form

$$A = \begin{pmatrix} a & b^T \\ b & C \end{pmatrix} = \begin{pmatrix} a & 0 \\ b & I \end{pmatrix} \begin{pmatrix} a^{-1} & 0 \\ 0 & C - ba^{-1}b^T \end{pmatrix} \begin{pmatrix} a & b^T \\ 0 & I \end{pmatrix}.$$

The `mk1_?spffrtx` routine performs $ncolm$ successive factorizations of the form

$$A = \begin{pmatrix} a & b^T \\ b & C \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ ba^{-1} & I \end{pmatrix} \begin{pmatrix} a & 0 \\ 0 & C - ba^{-1}b^T \end{pmatrix} \begin{pmatrix} 1 & (ba^{-1})^T \\ 0 & I \end{pmatrix}.$$

The approximate number of floating point operations performed by real flavors of these routines is $(1/6)*ncolm*(2*ncolm^2 - 6*ncolm*n + 3*ncolm + 6*n^2 - 6*n + 7)$.

The approximate number of floating point operations performed by complex flavors of these routines is $(1/3)*ncolm*(4*ncolm^2 - 12*ncolm*n + 9*ncolm + 12*n^2 - 18*n + 8)$.

Input Parameters

<i>ap</i>	Array, size at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the lower triangular part of the matrix A in packed storage (see Matrix Storage Schemes for <code>uplo = 'L'</code>).
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>ncolm</i>	The number of columns to factor, $ncolm \leq n$.
<i>work, work2</i>	Workspace arrays, size of each at least n .

Output Parameters

<i>ap</i>	Overwritten by the factor L . The first $ncolm$ diagonal elements of the input matrix A are replaced with the diagonal elements of D . The subdiagonal elements of the first $ncolm$ columns are replaced with the corresponding elements of L . The rest of the input array is updated as indicated in the Description section.
-----------	--

NOTE

Specifying $ncolm = n$ results in complete factorization $A = LDL^T$.

See Also

[mkl_progress](#)

Matrix Storage Schemes**Solving Systems of Linear Equations: LAPACK Computational Routines**

This section describes the LAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see [Routines for Matrix Factorization](#)). However, the factorization is not necessary if your system of equations has a triangular matrix.

?getrs

Solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides.

Syntax

```
lapack_int LAPACK_sgetrs (int matrix_layout , char trans , lapack_int n , lapack_int
nrhs , const float * a , lapack_int lda , const lapack_int * ipiv , float * b ,
lapack_int ldb );
```

```
lapack_int LAPACK_dgetrs (int matrix_layout , char trans , lapack_int n , lapack_int
nrhs , const double * a , lapack_int lda , const lapack_int * ipiv , double * b ,
lapack_int ldb );
```

```
lapack_int LAPACK_cgetrs (int matrix_layout , char trans , lapack_int n , lapack_int
nrhs , const lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv ,
lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACK_zgetrs (int matrix_layout , char trans , lapack_int n , lapack_int
nrhs , const lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- `mkl.h`

Description

The routine solves for X the following systems of linear equations:

$A * X = B$	if $trans = 'N'$,
$A^T * X = B$	if $trans = 'T'$,
$A^H * X = B$	if $trans = 'C'$ (for complex matrices only).

Before calling this routine, you must call [?getrf](#) to compute the LU factorization of A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>trans</i>	Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $A * X = B$ is solved for X . If <i>trans</i> = 'T', then $A^T * X = B$ is solved for X . If <i>trans</i> = 'C', then $A^H * X = B$ is solved for X .
<i>n</i>	The order of A ; the number of rows in B ($n \geq 0$).
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>a</i>	Array of size $\max(1, lda * n)$. The array <i>a</i> contains LU factorization of matrix A resulting from the call of ?getrf .
<i>b</i>	Array of size $\max(1, ldb * nrhs)$ for column major layout, and $\max(1, ldb * n)$ for row major layout. The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?getrf .

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
----------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n) \varepsilon P |L| |U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \varepsilon$$

where $\operatorname{cond}(A, x) = \|A^{-1}\|_\infty \|A\|_\infty \|x\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?gecon`.

To refine the solution and estimate the error, call `?gerfs`.

See Also

Matrix Storage Schemes

`?gbtrs`

Solves a system of linear equations with an LU-factored band coefficient matrix, with multiple right-hand sides.

Syntax

```
lapack_int LAPACKGE_sgbtrs (int matrix_layout , char trans , lapack_int n , lapack_int
kl , lapack_int ku , lapack_int nrhs , const float * ab , lapack_int ldab , const
lapack_int * ipiv , float * b , lapack_int ldb );
```

```
lapack_int LAPACKGE_dgbtrs (int matrix_layout , char trans , lapack_int n , lapack_int
kl , lapack_int ku , lapack_int nrhs , const double * ab , lapack_int ldab , const
lapack_int * ipiv , double * b , lapack_int ldb );
```

```
lapack_int LAPACKGE_cgbtrs (int matrix_layout , char trans , lapack_int n , lapack_int
kl , lapack_int ku , lapack_int nrhs , const lapack_complex_float * ab , lapack_int
ldab , const lapack_int * ipiv , lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKGE_zgbtrs (int matrix_layout , char trans , lapack_int n , lapack_int
kl , lapack_int ku , lapack_int nrhs , const lapack_complex_double * ab , lapack_int
ldab , const lapack_int * ipiv , lapack_complex_double * b , lapack_int ldb );
```

Include Files

- `mk1.h`

Description

The routine solves for X the following systems of linear equations:

$A * X = B$	if $trans = 'N'$,
$A^T * X = B$	if $trans = 'T'$,
$A^H * X = B$	if $trans = 'C'$ (for complex matrices only).

Here A is an LU -factored general band matrix of order n with kl non-zero subdiagonals and ku nonzero superdiagonals. Before calling this routine, call `?gbtrf` to compute the LU factorization of A .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>trans</code>	Must be 'N' or 'T' or 'C'.
<code>n</code>	The order of A ; the number of rows in B ; $n \geq 0$.
<code>kl</code>	The number of subdiagonals within the band of A ; $kl \geq 0$.
<code>ku</code>	The number of superdiagonals within the band of A ; $ku \geq 0$.
<code>nrhs</code>	The number of right-hand sides; $nrhs \geq 0$.
<code>ab</code>	Array <code>ab</code> size $\max(1, ldab*n)$ The array <code>ab</code> contains elements of the LU factors of the matrix A as returned by <code>gbtrf</code> .
<code>b</code>	Array <code>b</code> size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout. The array <code>b</code> contains the matrix B whose columns are the right-hand sides for the systems of equations.
<code>ldab</code>	The leading dimension of the array <code>ab</code> ; $ldab \geq 2*kl + ku + 1$.
<code>ldb</code>	The leading dimension of <code>b</code> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<code>ipiv</code>	Array, size at least $\max(1, n)$. The <code>ipiv</code> array, as returned by <code>gbtrf</code> .

Output Parameters

<code>b</code>	Overwritten by the solution matrix X .
----------------	--

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, parameter i had an illegal value.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kl + ku + 1)\varepsilon P|L||U|$$

$c(k)$ is a modest linear function of k , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kl + ku + 1) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $2n(ku + 2kl)$ for real flavors. The number of operations for complex flavors is 4 times greater. All these estimates assume that kl and ku are much less than $\min(m, n)$.

To estimate the condition number $\kappa_\infty(A)$, call [?gbcon](#).

To refine the solution and estimate the error, call [?gbrfs](#).

See Also

Matrix Storage Schemes

[?gttrs](#)

Solves a system of linear equations with a tridiagonal coefficient matrix using the LU factorization computed by [?gttrf](#).

Syntax

```
lapack_int LAPACKE_sgttrs (int matrix_layout , char trans , lapack_int n , lapack_int
nrhs , const float * dl , const float * d , const float * du , const float * du2 ,
const lapack_int * ipiv , float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_dgttrs (int matrix_layout , char trans , lapack_int n , lapack_int
nrhs , const double * dl , const double * d , const double * du , const double * du2 ,
const lapack_int * ipiv , double * b , lapack_int ldb );
```

```
lapack_int LAPACKE_cgttrs (int matrix_layout , char trans , lapack_int n , lapack_int
nrhs , const lapack_complex_float * dl , const lapack_complex_float * d , const
lapack_complex_float * du , const lapack_complex_float * du2 , const lapack_int *
ipiv , lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_zgttrs (int matrix_layout , char trans , lapack_int n , lapack_int
nrhs , const lapack_complex_double * dl , const lapack_complex_double * d , const
lapack_complex_double * du , const lapack_complex_double * du2 , const lapack_int *
ipiv , lapack_complex_double * b , lapack_int ldb );
```

Include Files

- `mkl.h`

Description

The routine solves for X the following systems of linear equations with multiple right hand sides:

$A * X = B$ if $trans = 'N'$,

$A^T * X = B$ if $trans = 'T'$,

$A^H * X = B$ if *trans* = 'C' (for complex matrices only).

Before calling this routine, you must call `?gttrf` to compute the *LU* factorization of *A*.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout for array <i>b</i> is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>trans</i>	Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $A * X = B$ is solved for <i>X</i> . If <i>trans</i> = 'T', then $A^T * X = B$ is solved for <i>X</i> . If <i>trans</i> = 'C', then $A^H * X = B$ is solved for <i>X</i> .
<i>n</i>	The order of <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, that is, the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>dl,d,du,du2</i>	Arrays: $dl(n-1)$, $d(n)$, $du(n-1)$, $du2(n-2)$. The array <i>dl</i> contains the $(n-1)$ multipliers that define the matrix <i>L</i> from the <i>LU</i> factorization of <i>A</i> . The array <i>d</i> contains the <i>n</i> diagonal elements of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i> . The array <i>du</i> contains the $(n-1)$ elements of the first superdiagonal of <i>U</i> . The array <i>du2</i> contains the $(n-2)$ elements of the second superdiagonal of <i>U</i> .
<i>b</i>	Array of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, n * ldb)$ for row major layout. Contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ipiv</i>	Array, size (n) . The <i>ipiv</i> array, as returned by <code>?gttrf</code> .

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n) \varepsilon P |L| |U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kl + ku + 1) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $7n$ (including n divisions) for real flavors and $34n$ (including $2n$ divisions) for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?gtcon](#).

To refine the solution and estimate the error, call [?gtrfs](#).

See Also

Matrix Storage Schemes

[?dttrs](#)

Solves a system of linear equations with a diagonally dominant tridiagonal coefficient matrix using the LU factorization computed by [?dttrfb](#).

Syntax

```
void sdttrs (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const float
* dl, const float * d, const float * du, float * b, const MKL_INT * ldb, MKL_INT *
info );

void ddttrs (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const double
* dl, const double * d, const double * du, double * b, const MKL_INT * ldb, MKL_INT *
info );

void cdttrs (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const
MKL_Complex8 * dl, const MKL_Complex8 * d, const MKL_Complex8 * du, MKL_Complex8 * b,
const MKL_INT * ldb, MKL_INT * info );

void zdttrs (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const
MKL_Complex16 * dl, const MKL_Complex16 * d, const MKL_Complex16 * du, MKL_Complex16 *
b, const MKL_INT * ldb, MKL_INT * info );
```

Include Files

- `mk1.h`

Description

The `?dtttrs` routine solves the following systems of linear equations with multiple right hand sides for X :

$$\begin{aligned} A * X &= B && \text{if } trans = 'N', \\ A^T * X &= B && \text{if } trans = 'T', \\ A^H * X &= B && \text{if } trans = 'C' \text{ (for complex matrices only).} \end{aligned}$$

Before calling this routine, call `?dtttrfb` to compute the factorization of A .

Input Parameters

<i>trans</i>	Must be 'N' or 'T' or 'C'. Indicates the form of the equations solved for X : If <i>trans</i> = 'N', then $A * X = B$. If <i>trans</i> = 'T', then $A^T * X = B$. If <i>trans</i> = 'C', then $A^H * X = B$.
<i>n</i>	The order of A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, that is, the number of columns in B ; $nrhs \geq 0$.
<i>dl, d, du</i>	Arrays: $dl(n-1)$, $d(n)$, $du(n-1)$. The array <i>dl</i> contains the $(n-1)$ multipliers that define the matrices L_1, L_2 from the factorization of A . The array <i>d</i> contains the n diagonal elements of the upper triangular matrix U from the factorization of A . The array <i>du</i> contains the $(n-1)$ elements of the superdiagonal of U .
<i>b</i>	Array of size $\max(1, ldb * nrhs)$. Contains the matrix B whose columns are the right-hand sides for the systems of equations.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

?potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

Syntax

```
lapack_int LAPACKE_spotrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const float * a , lapack_int lda , float * b , lapack_int ldb );

lapack_int LAPACKE_dpots (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const double * a , lapack_int lda , double * b , lapack_int ldb );
```

```
lapack_int LAPACKE_cpotrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_float * a , lapack_int lda , lapack_complex_float * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_zpotrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_double * a , lapack_int lda , lapack_complex_double * b ,
lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for X the system of linear equations $A * X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A , given the Cholesky factorization of A :

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo = 'U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo = 'L' \end{aligned}$$

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

Before calling this routine, you must call [?potrf](#) to compute the Cholesky factorization of A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', U is stored, where $A = U^T * U$ for real data, $A = U^H * U$ for complex data. If <i>uplo</i> = 'L', L is stored, where $A = L * L^T$ for real data, $A = L * L^H$ for complex data.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides ($nrhs \geq 0$).
<i>a</i>	Array A of size at least $\max(1, lda * n)$ The array <i>a</i> contains the factor U or L (see <i>uplo</i>) as returned by potrf .
<i>lda</i>	The leading dimension of <i>a</i> . $lda \geq \max(1, n)$.
<i>b</i>	The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations. The size of <i>b</i> must be at least $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout.
<i>ldb</i>	The leading dimension of <i>b</i> . $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

b Overwritten by the solution matrix X .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

If *uplo* = 'U', the computed solution for each right-hand side b is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for *uplo* = 'L'. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \| |A^{-1}| \|_\infty \| |A| \|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$. The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?pocon](#).

To refine the solution and estimate the error, call [?porfs](#).

See Also

Matrix Storage Schemes

[?pfttrs](#)

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix using the Rectangular Full Packed (RFP) format.

Syntax

```
lapack_int LAPACKE_spfttrs (int matrix_layout , char transr , char uplo , lapack_int n ,
lapack_int nrhs , const float * a , float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_dpfttrs (int matrix_layout , char transr , char uplo , lapack_int n ,
lapack_int nrhs , const double * a , double * b , lapack_int ldb );
```

```
lapack_int LAPACKE_cpfttrs (int matrix_layout , char transr , char uplo , lapack_int n ,
lapack_int nrhs , const lapack_complex_float * a , lapack_complex_float * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_zpftsr (int matrix_layout , char transr , char uplo , lapack_int n ,
lapack_int nrhs , const lapack_complex_double * a , lapack_complex_double * b ,
lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves a system of linear equations $A \cdot X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A using the Cholesky factorization of A :

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} & \text{if } uplo = 'U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} & \text{if } uplo = 'L' \end{aligned}$$

Before calling `?pftsr`, you must call `?pfttrf` to compute the Cholesky factorization of A . L stands for a lower triangular matrix and U for an upper triangular matrix.

The matrix A is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>transr</i>	Must be 'N', 'T' (for real data) or 'C' (for complex data). If <i>transr</i> = 'N', the untransposed factor of A is stored in RFP format. If <i>transr</i> = 'T', the transposed factor of A is stored in RFP format. If <i>transr</i> = 'C', the conjugate-transposed factor of A is stored in RFP format.
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', U is stored, where $A = U^T * U$ for real data, $A = U^H * U$ for complex data. If <i>uplo</i> = 'L', L is stored, where $A = L * L^T$ for real data, $A = L * L^H$ for complex data
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.
<i>a</i>	Array <i>a</i> of size $\max(1, n * (n + 1) / 2)$. The array <i>a</i> contains, in the RFP format, the factor U or L obtained by factorization of matrix A .
<i>b</i>	The array <i>b</i> of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout contains the matrix B whose columns are the right-hand sides for the systems of equations.

ldb The leading dimension of *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

b The solution matrix *X*.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

See Also

Matrix Storage Schemes

?pptrs

Solves a system of linear equations with a packed Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

Syntax

```
lapack_int LAPACKE_spptrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const float * ap , float * b , lapack_int ldb );

lapack_int LAPACKE_dpptrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const double * ap , double * b , lapack_int ldb );

lapack_int LAPACKE_cpptrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_float * ap , lapack_complex_float * b , lapack_int ldb );

lapack_int LAPACKE_zpptrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_double * ap , lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for *X* the system of linear equations $A * X = B$ with a packed symmetric positive-definite or, for complex data, Hermitian positive-definite matrix *A*, given the Cholesky factorization of *A*:

$$\begin{array}{ll} A = U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} & \text{if uplo='U'} \\ A = L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} & \text{if uplo='L'} \end{array}$$

where *L* is a lower triangular matrix and *U* is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix *B*.

Before calling this routine, you must call [?pptrf](#) to compute the Cholesky factorization of *A*.

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo Must be 'U' or 'L'.

Indicates how the input matrix A has been factored:

If $uplo = 'U'$, U is stored, where $A = U^T * U$ for real data, $A = U^H * U$ for complex data.

If $uplo = 'L'$, L is stored, where $A = L * L^T$ for real data, $A = L * L^H$ for complex data

n

The order of matrix A ; $n \geq 0$.

$nrhs$

The number of right-hand sides ($nrhs \geq 0$).

ap, b

The size of ap must be at least $\max(1, n(n+1)/2)$.

The array ap contains the factor U or L , as specified by $uplo$, in *packed storage* (see [Matrix Storage Schemes](#)).

b

The array b of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout contains the matrix B whose columns are the right-hand sides for the systems of equations.

ldb

The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

b

Overwritten by the solution matrix X .

Return Values

This function returns a value *info*.

If $info = 0$, the execution is successful.

If $info = -i$, parameter i had an illegal value.

Application Notes

If $uplo = 'U'$, the computed solution for each right-hand side b is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\epsilon \cdot |U^H| \cdot |U|$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

A similar estimate holds for $uplo = 'L'$.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \epsilon$$

where $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_{\infty}(A)$, call `?ppcon`.

To refine the solution and estimate the error, call `?pprfs`.

See Also

Matrix Storage Schemes

`?pbtrs`

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite band coefficient matrix.

Syntax

```
lapack_int LAPACKE_spbtrs (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , lapack_int nrhs , const float * ab , lapack_int ldab , float * b , lapack_int
ldb );
```

```
lapack_int LAPACKE_dpbtrs (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , lapack_int nrhs , const double * ab , lapack_int ldab , double * b , lapack_int
ldb );
```

```
lapack_int LAPACKE_cpbtrs (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , lapack_int nrhs , const lapack_complex_float * ab , lapack_int ldab ,
lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_zpbtrs (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , lapack_int nrhs , const lapack_complex_double * ab , lapack_int ldab ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- `mk1.h`

Description

The routine solves for real data a system of linear equations $A * X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite *band* matrix A , given the Cholesky factorization of A :

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo = 'U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo = 'L' \end{aligned}$$

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

Before calling this routine, you must call `?pbtrf` to compute the Cholesky factorization of A in the band storage form.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored:

If `uplo = 'U'`, U is stored in ab , where $A = U^T * U$ for real matrices and $A = U^H * U$ for complex matrices.

If `uplo = 'L'`, L is stored in ab , where $A = L * L^T$ for real matrices and $A = L * L^H$ for complex matrices.

n

The order of matrix A ; $n \geq 0$.

kd

The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.

$nrhs$

The number of right-hand sides; $nrhs \geq 0$.

ab

Array ab is of size $\max(1, ldab * n)$.

The array ab contains the Cholesky factor, as returned by the factorization routine, in *band storage* form.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

b

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

The size of b is at least $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout.

$ldab$

The leading dimension of the array ab ; $ldab \geq kd + 1$.

ldb

The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

b

Overwritten by the solution matrix X .

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, parameter i had an illegal value.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kd + 1)\epsilon P |U^H| |U| \text{ or } |E| \leq c(kd + 1)\epsilon P |L^H| |L|$$

$c(k)$ is a modest linear function of k , and ϵ is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kd + 1) \text{cond}(A, x) \epsilon$$

where $\text{cond}(A, x) = ||A^{-1}||_A ||x||_\infty / ||x||_\infty \leq ||A^{-1}||_\infty ||A||_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $4n*kd$ for real flavors and $16n*kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?pbcon`.

To refine the solution and estimate the error, call `?pbrfs`.

See Also

Matrix Storage Schemes

`?pttrs`

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal coefficient matrix using the factorization computed by `?pttrf`.

Syntax

```
lapack_int LAPACKE_spttrs( int matrix_layout, lapack_int n, lapack_int nrhs, const
float* d, const float* e, float* b, lapack_int ldb );
```

```
lapack_int LAPACKE_dpttrs( int matrix_layout, lapack_int n, lapack_int nrhs, const
double* d, const double* e, double* b, lapack_int ldb );
```

```
lapack_int LAPACKE_cpttrs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* d, const lapack_complex_float* e, lapack_complex_float* b, lapack_int
ldb );
```

```
lapack_int LAPACKE_zpttrs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* d, const lapack_complex_double* e, lapack_complex_double* b, lapack_int
ldb );
```

Include Files

- `mkl.h`

Description

The routine solves for X a system of linear equations $A^*X = B$ with a symmetric (Hermitian) positive-definite tridiagonal matrix A . Before calling this routine, call `?pttrf` to compute the $L^*D^*L^T$ or $U^T^*D^*U$ for real data and the $L^*D^*L^H$ or $U^H^*D^*U$ factorization of A for complex data.

Input Parameters

`matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

`uplo` Used for `cpttrs/zpttrs` only. Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and how A is factored:

If `uplo = 'U'`, the array `e` stores the conjugated values of the superdiagonal of U , and A is factored as $U^H^*D^*U$.

If `uplo = 'L'`, the array `e` stores the subdiagonal of L , and A is factored as $L^*D^*L^H$.

n	The order of A ; $n \geq 0$.
$nrhs$	The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.
d	Array, dimension (n) . Contains the diagonal elements of the diagonal matrix D from the factorization computed by ?pttrf .
e	Array e is of size $(n - 1)$. The array e contains the $(n - 1)$ sub-diagonal elements of the unit bidiagonal factor L or the conjugated values of the superdiagonal of U from the factorization computed by ?pttrf (see <i>uplo</i>).
e, b	The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The size of b is at least $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout.
ldb	The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

b	Overwritten by the solution matrix X .
-----	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = $-i$, parameter i had an illegal value.

See Also

Matrix Storage Schemes

[?sytrs](#)

Solves a system of linear equations with a UDU^T - or LDL^T -factored symmetric coefficient matrix.

Syntax

```
lapack_int LAPACKE_ssytrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const float * a , lapack_int lda , const lapack_int * ipiv , float * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_dsytrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const double * a , lapack_int lda , const lapack_int * ipiv , double * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_csytrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv ,
lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_zsytrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- `mkl.h`

Description

The routine solves for X the system of linear equations $A * X = B$ with a symmetric matrix A , given the Bunch-Kaufman factorization of A :

if $uplo = 'U'$, $A = U * D * U^T$
 if $uplo = 'L'$, $A = L * D * L^T$,

where U and L are upper and lower triangular matrices with unit diagonal and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply to this routine the factor U (or L) and the array $ipiv$ returned by the factorization routine [?sytrf](#).

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = U * D * U^T$. If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = L * D * L^T$.
<code>n</code>	The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides; $nrhs \geq 0$.
<code>ipiv</code>	Array, size at least $\max(1, n)$. The $ipiv$ array, as returned by ?sytrf .
<code>a</code>	The array a of size $\max(1, lda * n)$ contains the factor U or L (see $uplo$). .
<code>b</code>	The array b contains the matrix B whose columns are the right-hand sides for the system of equations. The size of b is at least $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout.
<code>lda</code>	The leading dimension of a ; $lda \geq \max(1, n)$.
<code>ldb</code>	The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<code>b</code>	Overwritten by the solution matrix X .
----------------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T \text{ or } |E| \leq c(n) \varepsilon P |L| |D| |U^T| P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x) \varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?sycon](#).

To refine the solution and estimate the error, call [?sytrfs](#).

See Also

Matrix Storage Schemes

[?sytrs_aa](#)

*Solves a system of linear equations $A * X = B$ with a symmetric matrix.*

```
lapack_int LAPACKESsytrs_aa (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const float * A, lapack_int lda, const lapack_int * ipiv, float * B, lapack_int
ldb);
```

```
lapack_int LAPACKESdytrs_aa (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const double * A, lapack_int lda, const lapack_int * ipiv, double * B, lapack_int
ldb);
```

```
lapack_int LAPACKESsytrs_aa (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const lapack_complex_float * A, lapack_int lda, const lapack_int * ipiv,
lapack_complex_float * B, lapack_int ldb);
```

```
lapack_int LAPACKESzsytrs_aa (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const lapack_complex_double * A, lapack_int lda, const lapack_int * ipiv,
lapack_complex_double * B, lapack_int ldb);
```


Description

?sytrs_aa solves a system of linear equations $A * X = B$ with a symmetric matrix A using the factorization $A = U * T * U^T$ or $A = L * T * L^T$ computed by ?sytrf_aa.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. <ul style="list-style-type: none"> = 'U': Upper triangular; the form is $A = U * T * U^T$. = 'L': Lower triangular; the form is $A = L * T * L^T$.
<i>n</i>	The order of the matrix A . $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; that is, the number of columns of the matrix B . $nrhs \geq 0$.
<i>A</i>	Array of size $\max(1, lda * n)$. Details of factors computed by ?sytrf_aa.
<i>lda</i>	The leading dimension of the array A .
<i>ipiv</i>	Array of size n . Details of the interchanges as computed by ?sytrf_aa.
<i>B</i>	Array of size $\max(1, ldb * nrhs)$. On entry, the right-hand side matrix B .
<i>ldb</i>	The leading dimension of the array B . $ldb \geq \max(1, n)$ for column-major layout and $ldb \geq nrhs$ for row-major layout.

Output Parameters

<i>B</i>	On exit, the solution matrix X .
----------	------------------------------------

Return Values

This function returns a value *info*.

= 0: Successful exit.

< 0: If *info* = -*i*, the *i*th argument had an illegal value.

?sytrs_rook

Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix.

Syntax

```
lapack_int LAPACKE_ssytrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const float * a, lapack_int lda, const lapack_int * ipiv, float * b, lapack_int
ldb);
```

```
lapack_int LAPACKE_dsytrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const double * a, lapack_int lda, const lapack_int * ipiv, double * b, lapack_int
ldb);
```

```
lapack_int LAPACKE_csytrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv,
lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zsytrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv,
lapack_complex_double * b, lapack_int ldb);
```

Include Files

- mkl.h

Description

The routine solves a system of linear equations $A \cdot X = B$ with a symmetric matrix A , using the factorization $A = U \cdot D \cdot U^T$ or $A = L \cdot D \cdot L^T$ computed by [?sytrf_rook](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout for array <i>b</i> is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the factorization is of the form $A = U \cdot D \cdot U^T$. If <i>uplo</i> = 'L', the factorization is of the form $A = L \cdot D \cdot L^T$.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sytrf_rook .
<i>a, b</i>	Arrays: <i>a</i> , size ($lda \cdot n$), <i>b</i> size ($ldb \cdot nrhs$). The array <i>a</i> contains the block diagonal matrix D and the multipliers used to obtain U or L as computed by ?sytrf_rook (see <i>uplo</i>). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the system of equations.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

`?hetrs`

Solves a system of linear equations with a UDU^T - or LDL^T -factored Hermitian coefficient matrix.

Syntax

```
lapack_int LAPACKE_chetrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv ,
lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_zhetrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- `mkl.h`

Description

The routine solves for X the system of linear equations $A * X = B$ with a Hermitian matrix A , given the Bunch-Kaufman factorization of A :

```
if uplo='U',           A = U*D*UH
if uplo='L',           A = L*D*LH,
```

where U and L are upper and lower triangular matrices with unit diagonal and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply to this routine the factor U (or L) and the array `ipiv` returned by the factorization routine `?hetrf`.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the array <code>a</code> stores the upper triangular factor U of the factorization $A = U * D * U^H$. If <code>uplo = 'L'</code> , the array <code>a</code> stores the lower triangular factor L of the factorization $A = L * D * L^H$.
<code>n</code>	The order of matrix A ; $n \geq 0$.

<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hetrf .
<i>a</i>	The array <i>aof</i> size $\max(1, lda*n)$ contains the factor <i>U</i> or <i>L</i> (see <i>uplo</i>).
<i>b</i>	The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The size of <i>b</i> is at least $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^H|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D||L^H|P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x)\varepsilon$$

where $\text{cond}(A, x) = \| |A^{-1}||A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$.

To estimate the condition number $\kappa_\infty(A)$, call [?hecon](#).

To refine the solution and estimate the error, call [?herfs](#).

See Also

Matrix Storage Schemes

`?hetrs_aa`

*BSolves a system of linear equations $A * X =$ with a complex Hermitian matrix.*

```
LAPACK_DECL lapack_int LAPACKE_chetrs_aa (int matrix_layout, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int *
ipiv, lapack_complex_float * b, lapack_int ldb );
```

```
LAPACK_DECL lapack_int LAPACKE_zhetrs_aa (int matrix_layout, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int *
ipiv, lapack_complex_double * b, lapack_int ldb );
```

Description

`?hetrs_aa` solves a system of linear equations $A * X = X$ with a complex Hermitian matrix A using the factorization $A = U * T * U^H$ or $A = L * T * L^H$ computed by `?hetrf_aa`.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. If <code>uplo = 'U'</code> : Upper triangular of the form $A = U * T * U^H$. If <code>uplo = 'L'</code> : Lower triangular of the form $A = L * T * L^H$.
<code>n</code>	The order of the matrix A . $n \geq 0$.
<code>nrhs</code>	The number of right hand sides: the number of columns of the matrix b . $nrhs \geq 0$.
<code>a</code>	Array of size $lda * n$. Details of factors computed by <code>?hetrf_aa</code> .
<code>lda</code>	The leading dimension of the array a . $lda \geq \max(1, n)$.
<code>ipiv</code>	Array of size (n) . Details of the interchanges as computed by <code>?hetrf_aa</code> .
<code>b</code>	Array of size $ldb * nrhs$. On entry, the right hand side matrix B .
<code>ldb</code>	The leading dimension of the array b . $ldb \geq \max(1, n)$.
<code>work</code>	See Syntax - Workspace . Array of size $(\max(1, lwork))$.
<code>lwork</code>	See Syntax - Workspace . $lwork \geq \max(1, 3 * n - 2)$.

Output Parameters

<code>b</code>	On exit, the solution matrix X .
----------------	------------------------------------

Return Values

This function returns a value `info`.

If `info = 0`: successful exit.

If $info < 0$: if $info = -i$, the i -th argument had an illegal value.

Syntax - Workspace

Use this interface if you want to explicitly provide the workspace array.

```
LAPACK_DECL lapack_int LAPACKE_chetrs_aa_work (int matrix_layout, char uplo, lapack_int
n, lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int *
ipiv, lapack_complex_float * b, lapack_int ldb, lapack_complex_float * work, lapack_int
lwork );
```

```
LAPACK_DECL lapack_int LAPACKE_zhetrs_aa_work (int matrix_layout, char uplo, lapack_int
n, lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int *
ipiv, lapack_complex_double * b, lapack_int ldb, lapack_complex_double * work,
lapack_int lwork );
```

?hetrs_rook

Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix.

Syntax

```
lapack_int LAPACKE_chetrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv,
lapack_complex_float * b, lapack_int ldb);
```

```
lapack_int LAPACKE_zhetrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv,
lapack_complex_double * b, lapack_int ldb);
```

Include Files

- mkl.h

Description

The routine solves for a system of linear equations $A^*X = B$ with a complex Hermitian matrix A using the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ computed by [?hetrf_rook](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout for array <i>b</i> is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', the factorization is of the form $A = U^*D^*U^H$. If <i>uplo</i> = 'L', the factorization is of the form $A = L^*D^*L^H$.
<i>n</i>	The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hetrf_rook .

a, *b*Arrays: *a* (*lda***n*), *b*(*ldb***nrhs*).

The array *a* contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* as computed by `?hetrf_rook` (see *uplo*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations.

*lda*The leading dimension of *a*; *lda* ≥ max(1, *n*).*ldb*

The leading dimension of *b*; *ldb* ≥ max(1, *n*) for column major layout and *ldb* ≥ *nrhs* for row major layout.

Output Parameters

*b*Overwritten by the solution matrix *X*.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?sytrs2

Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix.

Syntax

```
lapack_int LAPACKE_ssytrs2 (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const float * a , lapack_int lda , const lapack_int * ipiv , float * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_dsytrs2 (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const double * a , lapack_int lda , const lapack_int * ipiv , double * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_csytrs2 (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv ,
lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_zsytrs2 (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves a system of linear equations $A \cdot X = B$ with a symmetric matrix *A* using the factorization of *A*:

if *uplo*='U', $A = U \cdot D \cdot U^T$

if *uplo*='L', $A = L \cdot D \cdot L^T$

where

- U and L are upper and lower triangular matrices with unit diagonal
- D is a symmetric block-diagonal matrix.

The factorization is computed by `?sytrf`.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the array a stores the upper triangular factor U of the factorization $A = U * D * U^T$. If <code>uplo = 'L'</code> , the array a stores the lower triangular factor L of the factorization $A = L * D * L^T$.
<code>n</code>	The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides; $nrhs \geq 0$.
<code>a</code>	The array a of size $\max(1, lda * n)$ contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <code>?sytrf</code> .
<code>b</code>	The array b contains the right-hand side matrix B . The size of b is at least $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout.
<code>lda</code>	The leading dimension of a ; $lda \geq \max(1, n)$.
<code>ldb</code>	The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<code>ipiv</code>	Array of size n . The <code>ipiv</code> array contains details of the interchanges and the block structure of D as determined by <code>?sytrf</code> .

Output Parameters

<code>b</code>	Overwritten by the solution matrix X .
----------------	--

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, parameter i had an illegal value.

See Also

[?sytrf](#)

[Matrix Storage Schemes](#)

?hetrs2

Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix.

Syntax

```
lapack_int LAPACKE_chetrs2 (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv ,
lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_zhetrs2 (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves a system of linear equations $A \cdot X = B$ with a complex Hermitian matrix A using the factorization of A :

if $uplo='U'$, $A = U \cdot D \cdot U^H$

if $uplo='L'$, $A = L \cdot D \cdot L^H$

where

- U and L are upper and lower triangular matrices with unit diagonal
- D is a Hermitian block-diagonal matrix.

The factorization is computed by ?hetrf.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = U \cdot D \cdot U^H$. If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = L \cdot D \cdot L^H$.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>a</i>	The array a of size $\max(1, lda \cdot n)$ contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?hetrf.
<i>b</i>	The array b of size $\max(1, ldb \cdot nrhs)$ for column major layout and $\max(1, ldb \cdot n)$ for row major layout contains the right-hand side matrix B .

<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ipiv</i>	Array of size <i>n</i> . The <i>ipiv</i> array contains details of the interchanges and the block structure of <i>D</i> as determined by ?hetrf.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

See Also

[?hetrf](#)

Matrix Storage Schemes

?sytrs_3

*Solves a system of linear equations $A * X = B$ with a real or complex symmetric matrix.*

```
lapack_int LAPACK_essytrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const float * A, lapack_int lda, const float * e, const lapack_int * ipiv, float
* B, lapack_int ldb);
```

```
lapack_int LAPACK_dsytrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const double * A, lapack_int lda, const double * e, const lapack_int * ipiv,
double * B, lapack_int ldb);
```

```
lapack_int LAPACK_csytrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e,
const lapack_int * ipiv, lapack_complex_float * B, lapack_int ldb);
```

```
lapack_int LAPACK_zsytrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e,
const lapack_int * ipiv, lapack_complex_double * B, lapack_int ldb);
```

Description

?sytrs_3 solves a system of linear equations $A * X = B$ with a real or complex symmetric matrix *A* using the factorization computed by ?sytrf_rk: $A = P * U * D * (U^T)^*(P^T)$ or $A = P * L * D * (L^T)^*(P^T)$, where *U* (or *L*) is unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of *U* (or *L*), *P* is a permutation matrix, P^T is the transpose of *P*, and *D* is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This algorithm uses Level 3 BLAS.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies whether the details of the factorization are stored as an upper or lower triangular matrix:

- = 'U': Upper triangular; the form is $A = P*U*D*(U^T)*(P^T)$.
- = 'L': Lower triangular; the form is $A = P*L*D*(L^T)*(P^T)$.

*n*The order of the matrix *A*. $n \geq 0$.*nrhs*The number of right-hand sides; that is, the number of columns of the matrix *B*. $nrhs \geq 0$.*A*Array of size $\max(1, lda*n)$. Diagonal of the block diagonal matrix *D* and factors *U* or *L* as computed by `?sytrf_rk`:

- Only diagonal elements of the symmetric block diagonal matrix *D* on the diagonal of *A*; that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of *D* should be provided on entry in array *e*.

—and—

- If *uplo* = 'U', factor *U* in the superdiagonal part of *A*. If *uplo* = 'L', factor *L* in the subdiagonal part of *A*.

*lda*The leading dimension of the array *A*.*e*

Array of size *n*. On entry, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix *D* with 1-by-1 or 2-by-2 diagonal blocks. If *uplo* = 'U', $e(i) = D(i-1,i), i=2:N$, and *e*(1) is not referenced. If *uplo* = 'L', $e(i) = D(i+1,i), i=1:N-1$, and *e*(*n*) is not referenced.

NOTE For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e[k-1]$ is not referenced in both the *uplo* = 'U' and *uplo* = 'L' cases.

*ipiv*Array of size *n*. Details of the interchanges and the block structure of *D* as determined by `?sytrf_rk`.*B*On entry, the right-hand side matrix *B*.

The size of *B* is at least $\max(1, ldb*nrhs)$ for column-major layout and $\max(1, ldb*n)$ for row-major layout.

ldb

The leading dimension of the array *B*. $ldb \geq \max(1, n)$ for column-major layout and $ldb \geq nrhs$ for row-major layout.

Output Parameters

*B*On exit, the solution matrix *X*.

Return Values

This function returns a value *info*.

= 0: Successful exit.

< 0: If *info* = -*i*, the *i*th argument had an illegal value.

?hetrs_3

Solves a system of linear equations $A * X = B$ with a complex Hermitian matrix using the factorization computed by ?hetrf_rk.

```
lapack_int LAPACKE_chetrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e,
const lapack_int * ipiv, lapack_complex_float * B, lapack_int ldb);
```

```
lapack_int LAPACKE_zhetrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e,
const lapack_int * ipiv, lapack_complex_double * B, lapack_int ldb);
```

Description

?hetrs_3 solves a system of linear equations $A * X = B$ with a complex Hermitian matrix A using the factorization computed by ?hetrf_rk: $A = P * U * D * (U^H) * (P^T)$ or $A = P * L * D * (L^H) * (P^T)$, where U (or L) is unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This algorithm uses Level 3 BLAS.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies whether the details of the factorization are stored as an upper or lower triangular matrix: <ul style="list-style-type: none"> = 'U': Upper triangular; form is $A = P * U * D * (U^H) * (P^T)$. = 'L': Lower triangular; form is $A = P * L * D * (L^H) * (P^T)$.
<i>n</i>	The order of the matrix A. $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; that is, the number of columns in the matrix B. $nrhs \geq 0$.
<i>A</i>	Array of size $\max(1, lda * n)$. Diagonal of the block diagonal matrix D and factor U or L as computed by ?hetrf_rk: <ul style="list-style-type: none"> Only diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A; that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of D should be provided on entry in array <i>e</i>. If <i>uplo</i> = 'U', factor U in the superdiagonal part of A. If <i>uplo</i> = 'L', factor L in the subdiagonal part of A.
<i>lda</i>	The leading dimension of the array A.
<i>e</i>	Array of size <i>n</i> . On entry, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If <i>uplo</i> = 'U', $e(i) = D(i-1,i), i=2:N$, and $e(1)$ is not referenced. If <i>uplo</i> = 'L', $e(i) = D(i+1,i), i=1:N-1$, and $e(n)$ is not referenced.

NOTE For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e[k-1]$ is not referenced in both the *uplo* = 'U' and *uplo* = 'L' cases.

<i>ipiv</i>	Array of size (<i>n</i>). Details of the interchanges and the block structure of <i>D</i> as determined by <code>?hetrf_rk</code> .
<i>B</i>	On entry, the right-hand side matrix <i>B</i> . The size of <i>B</i> is at least $\max(1, \text{ldb} * \text{nrhs})$ for column-major layout and $\max(1, \text{ldb} * n)$ for row-major layout.
<i>ldb</i>	The leading dimension of the array <i>B</i> . $\text{ldb} \geq \max(1, n)$ for column-major layout and $\text{ldb} \geq \text{nrhs}$ for row-major layout.

Output Parameters

<i>B</i>	On exit, the solution matrix <i>X</i> .
----------	---

Return Values

This function returns a value *info*.

= 0: Successful exit.

< 0: If *info* = *-i*, the *i*th argument had an illegal value.

?sptsr

Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix using packed storage.

Syntax

```
lapack_int LAPACKE_ssptsr (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const float * ap , const lapack_int * ipiv , float * b , lapack_int ldb );

lapack_int LAPACKE_dsptsr (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const double * ap , const lapack_int * ipiv , double * b , lapack_int ldb );

lapack_int LAPACKE_csptsr (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_float * ap , const lapack_int * ipiv , lapack_complex_float
* b , lapack_int ldb );

lapack_int LAPACKE_zsptsr (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_double * ap , const lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for *X* the system of linear equations $A * X = B$ with a symmetric matrix *A*, given the Bunch-Kaufman factorization of *A*:

if <i>uplo</i> ='U',	$A = U * D * U^T$
if <i>uplo</i> ='L',	$A = L * D * L^T$,

where U and L are upper and lower *packed* triangular matrices with unit diagonal and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply the factor U (or L) and the array $ipiv$ returned by the factorization routine [?spturf](#).

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo</code> = 'U', the array <code>ap</code> stores the packed factor U of the factorization $A = U * D * U^T$. If <code>uplo</code> = 'L', the array <code>ap</code> stores the packed factor L of the factorization $A = L * D * L^T$.
<code>n</code>	The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides; $nrhs \geq 0$.
<code>ipiv</code>	Array, size at least $\max(1, n)$. The <code>ipiv</code> array, as returned by ?spturf .
<code>ap</code>	The dimension of array <code>ap</code> must be at least $\max(1, n(n+1)/2)$. The array <code>ap</code> contains the factor U or L , as specified by <code>uplo</code> , in <i>packed storage</i> (see Matrix Storage Schemes).
<code>b</code>	The array <code>b</code> contains the matrix B whose columns are the right-hand sides for the system of equations. The size of <code>b</code> is $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout.
<code>ldb</code>	The leading dimension of <code>b</code> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<code>b</code>	Overwritten by the solution matrix X .
----------------	--

Return Values

This function returns a value `info`.

If `info`=0, the execution is successful.

If `info` = $-i$, parameter i had an illegal value.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T \text{ or } |E| \leq c(n) \varepsilon P |L| |D| |L^T| P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \varepsilon$$

where $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?spcon](#).

To refine the solution and estimate the error, call [?sprfs](#).

See Also

Matrix Storage Schemes

[?hptrs](#)

Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix using packed storage.

Syntax

```
lapack_int LAPACKE_chptrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_float * ap , const lapack_int * ipiv , lapack_complex_float
* b , lapack_int ldb );
```

```
lapack_int LAPACKE_zhptrs (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , const lapack_complex_double * ap , const lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- `mkl.h`

Description

The routine solves for X the system of linear equations $A * X = B$ with a Hermitian matrix A , given the Bunch-Kaufman factorization of A :

```
if uplo='U',           A = U*D*UH
if uplo='L',           A = L*D*LH,
```

where U and L are upper and lower *packed* triangular matrices with unit diagonal and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

You must supply to this routine the arrays ap (containing U or L) and $ipiv$ in the form returned by the factorization routine [?hptrf](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the packed factor <i>U</i> of the factorization $A = U^*D^*U^H$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the packed factor <i>L</i> of the factorization $A = L^*D^*L^H$.
<i>n</i>	The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hptrf .
<i>ap</i>	The dimension of array <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> , in <i>packed storage</i> (see Matrix Storage Schemes).
<i>b</i>	The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The size of <i>b</i> is $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^H| P^T \text{ or } |E| \leq c(n) \varepsilon P |L| |D| |L^H| P^T$$

$c(n)$ is a modest linear function of *n*, and ε is the machine precision.

If x_0 is the true solution, the computed solution *x* satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \varepsilon$$

where $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?hpcon](#).

To refine the solution and estimate the error, call [?hprfs](#).

See Also

Matrix Storage Schemes

[?trtrs](#)

Solves a system of linear equations with a triangular coefficient matrix, with multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_strtrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int nrhs , const float * a , lapack_int lda , float * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_dtrtrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int nrhs , const double * a , lapack_int lda , double * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_ctrtrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int nrhs , const lapack_complex_float * a , lapack_int lda ,
lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_ztrtrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int nrhs , const lapack_complex_double * a , lapack_int lda ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- `mkl.h`

Description

The routine solves for X the following systems of linear equations with a triangular matrix A , with multiple right-hand sides stored in B :

$A * X = B$	if $trans = 'N'$,
$A^T * X = B$	if $trans = 'T'$,
$A^H * X = B$	if $trans = 'C'$ (for complex matrices only).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether <i>A</i> is upper or lower triangular: If <i>uplo</i> = 'U', then <i>A</i> is upper triangular. If <i>uplo</i> = 'L', then <i>A</i> is lower triangular.
<i>trans</i>	Must be 'N' or 'T' or 'C'. If <i>trans</i> = 'N', then $A * X = B$ is solved for <i>X</i> . If <i>trans</i> = 'T', then $A^T * X = B$ is solved for <i>X</i> . If <i>trans</i> = 'C', then $A^H * X = B$ is solved for <i>X</i> .
<i>diag</i>	Must be 'N' or 'U'. If <i>diag</i> = 'N', then <i>A</i> is not a unit triangular matrix. If <i>diag</i> = 'U', then <i>A</i> is unit triangular: diagonal elements of <i>A</i> are assumed to be 1 and not referenced in the array <i>a</i> .
<i>n</i>	The order of <i>A</i> ; the number of rows in <i>B</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>a</i>	The array <i>a</i> contains the matrix <i>A</i> . The size of <i>a</i> is $\max(1, lda * n)$.
<i>b</i>	The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The size of <i>b</i> is $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |A|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x) \varepsilon \text{ provided } c(n) \operatorname{cond}(A, x) \varepsilon < 1$$

where $\operatorname{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call `?trcon`.

To estimate the error in the solution, call `?trrfs`.

See Also

Matrix Storage Schemes

?tpttrs

Solves a system of linear equations with a packed triangular coefficient matrix, with multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_stpttrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int nrhs , const float * ap , float * b , lapack_int ldb );

lapack_int LAPACKE_dtpttrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int nrhs , const double * ap , double * b , lapack_int ldb );

lapack_int LAPACKE_ctpttrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int nrhs , const lapack_complex_float * ap , lapack_complex_float
* b , lapack_int ldb );

lapack_int LAPACKE_ztpttrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int nrhs , const lapack_complex_double * ap ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for X the following systems of linear equations with a packed triangular matrix A , with multiple right-hand sides stored in B :

$A * X = B$ if *trans* = 'N',
 $A^T * X = B$ if *trans* = 'T',
 $A^H * X = B$ if *trans* = 'C' (for complex matrices only).

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo Must be 'U' or 'L'.
 Indicates whether *A* is upper or lower triangular:
 If *uplo* = 'U', then *A* is upper triangular.
 If *uplo* = 'L', then *A* is lower triangular.

trans Must be 'N' or 'T' or 'C'.
 If *trans* = 'N', then $A * X = B$ is solved for *X*.
 If *trans* = 'T', then $A^T * X = B$ is solved for *X*.
 If *trans* = 'C', then $A^H * X = B$ is solved for *X*.

diag Must be 'N' or 'U'.
 If *diag* = 'N', then *A* is not a unit triangular matrix.
 If *diag* = 'U', then *A* is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array *ap*.

n The order of *A*; the number of rows in *B*; $n \geq 0$.

nrhs The number of right-hand sides; $nrhs \geq 0$.

ap The dimension of array *ap* must be at least $\max(1, n(n+1)/2)$. The array *ap* contains the matrix *A* in *packed storage* (see [Matrix Storage Schemes](#)).

b The array *b* contains the matrix *B* whose columns are the right-hand sides for the system of equations.
 The size of *b* is $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout.

ldb The leading dimension of *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

b Overwritten by the solution matrix *X*.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |A|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x)\varepsilon \text{ provided } c(n) \text{cond}(A, x)\varepsilon < 1$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?tpcon](#).

To estimate the error in the solution, call [?tprfs](#).

See Also

Matrix Storage Schemes

[?tbtrs](#)

Solves a system of linear equations with a band triangular coefficient matrix, with multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_stbtrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int kd , lapack_int nrhs , const float * ab , lapack_int ldab ,
float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_dtbtrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int kd , lapack_int nrhs , const double * ab , lapack_int ldab ,
double * b , lapack_int ldb );
```

```
lapack_int LAPACKE_ctbtrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int kd , lapack_int nrhs , const lapack_complex_float * ab ,
lapack_int ldab , lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_ztbtrs (int matrix_layout , char uplo , char trans , char diag ,
lapack_int n , lapack_int kd , lapack_int nrhs , const lapack_complex_double * ab ,
lapack_int ldab , lapack_complex_double * b , lapack_int ldb );
```

Include Files

- `mk1.h`

Description

The routine solves for X the following systems of linear equations with a band triangular matrix A , with multiple right-hand sides stored in B :

$$\begin{aligned} A * X &= B & \text{if } trans = 'N', \\ A^T * X &= B & \text{if } trans = 'T', \\ A^H * X &= B & \text{if } trans = 'C' \text{ (for complex matrices only).} \end{aligned}$$

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>trans</i>	Must be 'N' or 'T' or 'C'. If <i>trans</i> = 'N', then $A * X = B$ is solved for X . If <i>trans</i> = 'T', then $A^T * X = B$ is solved for X . If <i>trans</i> = 'C', then $A^H * X = B$ is solved for X .
<i>diag</i>	Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ab</i> .
<i>n</i>	The order of A ; the number of rows in B ; $n \geq 0$.
<i>kd</i>	The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>ab</i>	The array <i>ab</i> contains the matrix A in <i>band storage</i> form. The size of <i>ab</i> must be $\max(1, ldab * n)$
<i>b</i>	The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations. The size of <i>b</i> is $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout.
<i>ldab</i>	The leading dimension of <i>ab</i> ; $ldab \geq kd + 1$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon|A|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x)\varepsilon \text{ provided } c(n) \text{cond}(A, x)\varepsilon < 1$$

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector *b* is $2n*kd$ for real flavors and $8n*kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?tbcon](#).

To estimate the error in the solution, call [?tbrfs](#).

See Also

Matrix Storage Schemes

Estimating the Condition Number: LAPACK Computational Routines

This section describes the LAPACK routines for estimating the *condition number* of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations (see [Error Analysis](#)). Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

[?gecon](#)

Estimates the reciprocal of the condition number of a general matrix in the 1-norm or the infinity-norm.

Syntax

```
lapack_int LAPACKGE_sgecon( int matrix_layout, char norm, lapack_int n, const float* a,
lapack_int lda, float anorm, float* rcond );
```

```
lapack_int LAPACKGE_dgecon( int matrix_layout, char norm, lapack_int n, const double* a,
lapack_int lda, double anorm, double* rcond );
```

```
lapack_int LAPACKGE_cgecon( int matrix_layout, char norm, lapack_int n, const
lapack_complex_float* a, lapack_int lda, float anorm, float* rcond );
```

```
lapack_int LAPACKGE_zgecon( int matrix_layout, char norm, lapack_int n, const
lapack_complex_double* a, lapack_int lda, double anorm, double* rcond );
```

Include Files

- `mk1.h`

Description

The routine estimates the reciprocal of the condition number of a general matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?getrf` to compute the *LU* factorization of A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>norm</i>	Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	The array <i>a</i> contains the <i>LU</i> -factored matrix A , as returned by <code>?getrf</code> .
<i>anorm</i>	The norm of the <i>original</i> matrix A (see Description).
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.

Output Parameters

<i>rcond</i>	An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
--------------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$ or $A^H*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2*n^2$ floating-point operations for real flavors and $8*n^2$ for complex flavors.

See Also

Matrix Storage Schemes

`?gbcon`

Estimates the reciprocal of the condition number of a band matrix in the 1-norm or the infinity-norm.

Syntax

```
lapack_int LAPACKE_sgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl,
lapack_int ku, const float* ab, lapack_int ldab, const lapack_int* ipiv, float anorm,
float* rcond );

lapack_int LAPACKE_dgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl,
lapack_int ku, const double* ab, lapack_int ldab, const lapack_int* ipiv, double anorm,
double* rcond );

lapack_int LAPACKE_cgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl,
lapack_int ku, const lapack_complex_float* ab, lapack_int ldab, const lapack_int* ipiv,
float anorm, float* rcond );

lapack_int LAPACKE_zgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl,
lapack_int ku, const lapack_complex_double* ab, lapack_int ldab, const lapack_int*
ipiv, double anorm, double* rcond );
```

Include Files

- `mk1.h`

Description

The routine estimates the reciprocal of the condition number of a general band matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute $anorm$ (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?gbtrf` to compute the LU factorization of A .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>norm</code>	Must be '1' or 'O' or 'I'.

If *norm* = '1' or 'O', then the routine estimates the condition number of matrix *A* in 1-norm.

If *norm* = 'I', then the routine estimates the condition number of matrix *A* in infinity-norm.

<i>n</i>	The order of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	The number of subdiagonals within the band of <i>A</i> ; $k \geq 0$.
<i>ku</i>	The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>ldab</i>	The leading dimension of the array <i>ab</i> . ($ldab \geq 2*kl + ku + 1$).
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?gbtrf .
<i>ab</i>	The array <i>ab</i> of size $\max(1, ldab*n)$ contains the factored band matrix <i>A</i> , as returned by ?gbtrf .
<i>anorm</i>	The norm of the <i>original</i> matrix <i>A</i> (see Description).

Output Parameters

<i>rcond</i>	An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
--------------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A*x = b$ or $A^H*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n(ku + 2kl)$ floating-point operations for real flavors and $8n(ku + 2kl)$ for complex flavors.

See Also

Matrix Storage Schemes

[?gtcon](#)

Estimates the reciprocal of the condition number of a tridiagonal matrix.

Syntax

```
lapack_int LAPACKE_sgtcon( char norm, lapack_int n, const float* dl, const float* d,
const float* du, const float* du2, const lapack_int* ipiv, float anorm, float* rcond );

lapack_int LAPACKE_dgtcon( char norm, lapack_int n, const double* dl, const double* d,
const double* du, const double* du2, const lapack_int* ipiv, double anorm, double*
rcond );
```

```
lapack_int LAPACKE_cgtcon( char norm, lapack_int n, const lapack_complex_float* dl,
const lapack_complex_float* d, const lapack_complex_float* du, const
lapack_complex_float* du2, const lapack_int* ipiv, float anorm, float* rcond );

lapack_int LAPACKE_zgtcon( char norm, lapack_int n, const lapack_complex_double* dl,
const lapack_complex_double* d, const lapack_complex_double* du, const
lapack_complex_double* du2, const lapack_int* ipiv, double anorm, double* rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a real or complex tridiagonal matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?gttrf](#) to compute the LU factorization of A .

Input Parameters

<i>norm</i>	Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>dl,d,du,du2</i>	Arrays: $dl(n-1)$, $d(n)$, $du(n-1)$, $du2(n-2)$. The array <i>dl</i> contains the $(n-1)$ multipliers that define the matrix L from the LU factorization of A as computed by ?gttrf . The array <i>d</i> contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A . The array <i>du</i> contains the $(n-1)$ elements of the first superdiagonal of U . The array <i>du2</i> contains the $(n-2)$ elements of the second superdiagonal of U .
<i>ipiv</i>	Array, size (n) . The array of pivot indices, as returned by ?gttrf .
<i>anorm</i>	The norm of the <i>original</i> matrix A (see <i>Description</i>).

Output Parameters

rcond

An estimate of the reciprocal of the condition number. The routine sets *rcond*=0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $A \cdot x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pocon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix.

Syntax

```
lapack_int LAPACKE_spocon( int matrix_layout, char uplo, lapack_int n, const float* a,
lapack_int lda, float anorm, float* rcond );

lapack_int LAPACKE_dpocon( int matrix_layout, char uplo, lapack_int n, const double* a,
lapack_int lda, double anorm, double* rcond );

lapack_int LAPACKE_cpocon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* a, lapack_int lda, float anorm, float* rcond );

lapack_int LAPACKE_zpocon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* a, lapack_int lda, double anorm, double* rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix *A*:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)

- call `?potrf` to compute the Cholesky factorization of A .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo</code> = 'U', A is factored as $A = U^T * U$ for real flavors or $A = U^H * U$ for complex flavors, and U is stored. If <code>uplo</code> = 'L', A is factored as $A = L * L^T$ for real flavors or $A = L * L^H$ for complex flavors, and L is stored.
<code>n</code>	The order of the matrix A ; $n \geq 0$.
<code>a</code>	The array <code>a</code> of size $\max(1, lda * n)$ contains the factored matrix A , as returned by <code>?potrf</code> .
<code>lda</code>	The leading dimension of <code>a</code> ; $lda \geq \max(1, n)$.
<code>anorm</code>	The norm of the <i>original</i> matrix A (see <i>Description</i>).

Output Parameters

<code>rcond</code>	An estimate of the reciprocal of the condition number. The routine sets <code>rcond</code> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <code>rcond</code> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
--------------------	--

Return Values

This function returns a value `info`.

If `info` = 0, the execution is successful.

If `info` = $-i$, parameter i had an illegal value.

Application Notes

The computed `rcond` is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A * x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

`?ppcon`

Estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix.

Syntax

```
lapack_int LAPACKE_sppcon( int matrix_layout, char uplo, lapack_int n, const float* ap,
float anorm, float* rcond );
```

```
lapack_int LAPACKE_dppcon( int matrix_layout, char uplo, lapack_int n, const double*
ap, double anorm, double* rcond );
```

```
lapack_int LAPACKE_cppcon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* ap, float anorm, float* rcond );
```

```
lapack_int LAPACKE_zppcon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* ap, double anorm, double* rcond );
```

Include Files

- `mkl.h`

Description

The routine estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

An estimate is obtained for $\|A^{-1}\|_1$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?pptrf` to compute the Cholesky factorization of A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', A is factored as $A = U^T * U$ for real flavors or $A = U^H * U$ for complex flavors, and U is stored. If <i>uplo</i> = 'L', A is factored as $A = L * L^T$ for real flavors or $A = L * L^H$ for complex flavors, and L is stored.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>ap</i>	The array <i>ap</i> contains the packed factored matrix A , as returned by <code>?pptrf</code> . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$.
<i>anorm</i>	The norm of the <i>original</i> matrix A (see <i>Description</i>).

Output Parameters

<i>rcond</i>	An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
--------------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?pbcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix.

Syntax

```
lapack_int LAPACKE_spbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const float* ab, lapack_int ldab, float anorm, float* rcond );
```

```
lapack_int LAPACKE_dpbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const double* ab, lapack_int ldab, double anorm, double* rcond );
```

```
lapack_int LAPACKE_cpbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const lapack_complex_float* ab, lapack_int ldab, float anorm, float* rcond );
```

```
lapack_int LAPACKE_zpbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const lapack_complex_double* ab, lapack_int ldab, double anorm, double* rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix *A*:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric or Hermitian, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as *rcond* = $1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?pbtrf](#) to compute the Cholesky factorization of *A*.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', <i>A</i> is factored as $A = U^T * U$ for real flavors or $A = U^H * U$ for complex flavors, and <i>U</i> is stored. If <i>uplo</i> = 'L', <i>A</i> is factored as $A = L * L^T$ for real flavors or $A = L * L^H$ for complex flavors, and <i>L</i> is stored.
<i>n</i>	The order of the matrix <i>A</i> ; $n \geq 0$.
<i>kd</i>	The number of superdiagonals or subdiagonals in the matrix <i>A</i> ; $kd \geq 0$.
<i>ldab</i>	The leading dimension of the array <i>ab</i> . ($ldab \geq kd + 1$).
<i>ab</i>	The array <i>ab</i> of size $\max(1, ldab * n)$ contains the factored matrix <i>A</i> in band form, as returned by ?pbtrf .
<i>anorm</i>	The norm of the <i>original</i> matrix <i>A</i> (see <i>Description</i>).

Output Parameters

<i>rcond</i>	An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
--------------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $A * x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4 * n * (kd + 1)$ floating-point operations for real flavors and $16 * n * (kd + 1)$ for complex flavors.

See Also

Matrix Storage Schemes

[?ptcon](#)

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

```
lapack_int LAPACKE_sptcon( lapack_int n, const float* d, const float* e, float anorm, float* rcond );
```

```
lapack_int LAPACKE_dptcon( lapack_int n, const double* d, const double* e, double anorm, double* rcond );
```



```
lapack_int LAPACKE_cptcon( lapack_int n, const float* d, const lapack_complex_float* e,
float anorm, float* rcond );
```

```
lapack_int LAPACKE_zptcon( lapack_int n, const double* d, const lapack_complex_double*
e, double anorm, double* rcond );
```

Include Files

- mkl.h

Description

The routine computes the reciprocal of the condition number (in the 1-norm) of a real symmetric or complex Hermitian positive-definite tridiagonal matrix using the factorization $A = L^*D^*L^T$ for real flavors and $A = L^*D^*L^H$ for complex flavors or $A = U^T*D^*U$ for real flavors and $A = U^H*D^*U$ for complex flavors computed by [?pttrf](#) :

$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$ (since A is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

The norm $\|A^{-1}\|_1$ is computed by a direct method, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:

- compute *anorm* as $\|A\|_1 = \max_j \sum_i |a_{ij}|$
- call [?pttrf](#) to compute the factorization of A .

Input Parameters

<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>d</i>	Arrays, dimension (<i>n</i>). The array <i>d</i> contains the <i>n</i> diagonal elements of the diagonal matrix D from the factorization of A , as computed by ?pttrf ;
<i>e</i>	Array, size (<i>n</i> - 1). Contains off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by ?pttrf .
<i>anorm</i>	The 1- norm of the <i>original</i> matrix A (see <i>Description</i>).

Output Parameters

<i>rcond</i>	An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
--------------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A \cdot x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4 \cdot n(kd + 1)$ floating-point operations for real flavors and $16 \cdot n(kd + 1)$ for complex flavors.

?ssycon

Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

```
lapack_int LAPACKE_ssycon( int matrix_layout, char uplo, lapack_int n, const float* a,
lapack_int lda, const lapack_int* ipiv, float anorm, float* rcond );

lapack_int LAPACKE_dsycon( int matrix_layout, char uplo, lapack_int n, const double* a,
lapack_int lda, const lapack_int* ipiv, double anorm, double* rcond );

lapack_int LAPACKE_csycon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* a, lapack_int lda, const lapack_int* ipiv, float anorm, float*
rcond );

lapack_int LAPACKE_zsycon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* a, lapack_int lda, const lapack_int* ipiv, double anorm, double*
rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a symmetric matrix *A*:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

An estimate is obtained for $\|A^{-1}\|_1$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?sytrf](#) to compute the factorization of *A*.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular factor <i>U</i> of the factorization $A = U \cdot D \cdot U^T$.

If `uplo = 'L'`, the array `a` stores the lower triangular factor L of the factorization $A = L * D * L^T$.

`n`

The order of matrix A ; $n \geq 0$.

`a`

The array `a` of size $\max(1, lda * n)$ contains the factored matrix A , as returned by `?sytrf`.

`lda`

The leading dimension of `a`; $lda \geq \max(1, n)$.

`ipiv`

Array, size at least $\max(1, n)$.

The array `ipiv`, as returned by `?sytrf`.

`anorm`

The norm of the *original* matrix A (see *Description*).

Output Parameters

`rcond`

An estimate of the reciprocal of the condition number. The routine sets `rcond = 0` if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime `rcond` is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, parameter `i` had an illegal value.

Application Notes

The computed `rcond` is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A * x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

`?sycon_3`

Estimates the reciprocal of the condition number (in the 1-norm) of a real or complex symmetric matrix A using the factorization computed by `?sytrf_rk`.

```
lapack_int LAPACKE_ssycon_3 (int matrix_layout, char uplo, lapack_int n, const float *
A, lapack_int lda, const float * e, const lapack_int * ipiv, float anorm, float *
rcond);
```

```
lapack_int LAPACKE_dsycon_3 (int matrix_layout, char uplo, lapack_int n, const double *
A, lapack_int lda, const double * e, const lapack_int * ipiv, double anorm, double *
rcond);
```

```
lapack_int LAPACKE_csycon_3 (int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e, const
lapack_int * ipiv, float anorm, float * rcond);
```

```
lapack_int LAPACKE_zsycon_3 (int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e, const
lapack_int * ipiv, double anorm, double * rcond);
```

Description

`?sycon_3` estimates the reciprocal of the condition number (in the 1-norm) of a real or complex symmetric matrix A using the factorization computed by `?sytrf_rk`. $A = P*U*D*(U^T)*(P^T)$ or $A = P*L*D*(L^T)*(P^T)$, where U (or L) is unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of U (or L), P is a permutation matrix, P^T is the transpose of P , and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

An estimate is obtained for $\text{norm}(\text{inv}(A))$, and the reciprocal of the condition number is computed as $rcond = 1 / (anorm * \text{norm}(\text{inv}(A)))$.

This routine uses BLAS3 solver `?sytrs_3`.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Specifies whether the details of the factorization are stored as an upper or lower triangular matrix: <ul style="list-style-type: none"> <code>'U'</code>: Upper triangular. The form is $A = P*U*D*(U^T)*(P^T)$. <code>'L'</code>: Lower triangular. The form is $A = P*L*D*(L^T)*(P^T)$.
<code>n</code>	The order of the matrix A . $n \geq 0$.
<code>A</code>	Array of size $\max(1, lda*n)$. Diagonal of the block diagonal matrix D and factors U or L as computed by <code>?sytrf_rk</code> : <ul style="list-style-type: none"> Only diagonal elements of the symmetric block diagonal matrix D on the diagonal of A; that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of D should be provided on entry in array <code>e</code>. —and— If <code>uplo = 'U'</code>, factor U in the superdiagonal part of A. If <code>uplo = 'L'</code>, factor L in the subdiagonal part of A.
<code>lda</code>	The leading dimension of the array <code>A</code> .
<code>e</code>	Array of size <code>n</code> . On entry, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If <code>uplo = 'U'</code> , $e(i) = D(i-1,i)$, $i=2:N$, and $e(1)$ is not referenced. If <code>uplo = 'L'</code> , $e(i) = D(i+1,i)$, $i=1:N-1$, and $e(n)$ is not referenced.
<hr/> <p>NOTE For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element <code>e[k-1]</code> is not referenced in both the <code>uplo = 'U'</code> and <code>uplo = 'L'</code> cases.</p> <hr/>	
<code>ipiv</code>	Array of size <code>n</code> . Details of the interchanges and the block structure of D as determined by <code>?sytrf_rk</code> .
<code>anorm</code>	The 1-norm of the original matrix A .

Output Parameters

rcond The reciprocal of the condition number of the matrix *A*, computed as *rcond* = $1/(anorm * AINVNM)$, where *AINVNM* is an estimate of the 1-norm of *inv(A)* computed in this routine.

Return Values

This function returns a value *info*.

= 0: Successful exit.

< 0: If *info* = $-i$, the i^{th} argument had an illegal value.

?hecon

Estimates the reciprocal of the condition number of a Hermitian matrix.

Syntax

```
lapack_int LAPACKE_checon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* a, lapack_int lda, const lapack_int* ipiv, float anorm, float*
rcond );
```

```
lapack_int LAPACKE_zhecon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* a, lapack_int lda, const lapack_int* ipiv, double anorm, double*
rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix *A*:

$\kappa_1(A) = ||A||_1 ||A^{-1}||_1$ (since *A* is Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call [?hetrf](#) to compute the factorization of *A*.

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', the array *a* stores the upper triangular factor *U* of the factorization $A = U^* D U^H$.

If *uplo* = 'L', the array *a* stores the lower triangular factor *L* of the factorization $A = L^* D L^H$.

n The order of matrix *A*; $n \geq 0$.

<i>a</i>	The array <i>a</i> of size $\max(1, lda \cdot n)$ contains the factored matrix <i>A</i> , as returned by ?hetrf .
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The array <i>ipiv</i> , as returned by ?hetrf .
<i>anorm</i>	The norm of the <i>original</i> matrix <i>A</i> (see <i>Description</i>).

Output Parameters

<i>rcond</i>	An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
--------------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $A \cdot x = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

See Also

Matrix Storage Schemes

[?hecon_3](#)

Estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian matrix A.

```
lapack_int LAPACKE_checon_3 (int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e, const
lapack_int * ipiv, float anorm, float * rcond);
```

```
lapack_int LAPACKE_zhecon_3 (int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e, const
lapack_int * ipiv, double anorm, double * rcond);
```

Description

[?hecon_3](#) estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian matrix *A* using the factorization computed by [?hetrf_rk](#): $A = P \cdot U \cdot D \cdot (U^H)^* \cdot (P^T)$ or $A = P \cdot L \cdot D \cdot (L^H)^* \cdot (P^T)$, where *U* (or *L*) is unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of *U* (or *L*), *P* is a permutation matrix, P^T is the transpose of *P*, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. An estimate is obtained for $\text{norm}(\text{inv}(A))$, and the reciprocal of the condition number is computed as $rcond = 1 / (anorm \cdot \text{norm}(\text{inv}(A)))$.

This routine uses BLAS3 solver [?hetrs_3](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>uplo</i>	Specifies whether the details of the factorization are stored as an upper or lower triangular matrix: = 'U': Upper triangular, form is $A = P*U*D*(U^H)*(P^T)$; = 'L': Lower triangular, form is $A = P*L*D*(L^H)*(P^T)$.
<i>n</i>	The order of the matrix A. $n \geq 0$.
<i>A</i>	<p>Array of size $\max(1, lda*n)$. Diagonal of the block diagonal matrix D and factor U or L as computed by <code>?hetrf_rk</code>:</p> <ul style="list-style-type: none"> Only diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A—that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of D must be provided on entry in array <i>e</i>. <p>—and—</p> <ul style="list-style-type: none"> If <i>uplo</i> = 'U', factor U in the superdiagonal part of A. If <i>uplo</i> = 'L', factor L in the subdiagonal part of A.
<i>lda</i>	The leading dimension of the array <i>A</i> .
<i>e</i>	<p>Array of size <i>n</i>. On entry, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If <i>uplo</i> = 'U', $e(i) = D(i-1, i), i=2:N$, and <i>e</i>(1) is not referenced. If <i>uplo</i> = 'L', $e(i) = D(i+1, i), i=1:N-1$, and <i>e</i>(<i>n</i>) is not referenced.</p>
<hr/> <p>NOTE For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e[k-1]$ is not referenced in both the <i>uplo</i> = 'U' and <i>uplo</i> = 'L' cases.</p> <hr/>	
<i>ipiv</i>	Array of size <i>n</i> . Details of the interchanges and the block structure of D as determined by <code>?hetrf_rk</code> .
<i>anorm</i>	The 1-norm of the original matrix A.

Output Parameters

<i>rcond</i>	The reciprocal of the condition number of the matrix A, computed as $rcond = 1/(anorm * AINVNM)$, where <i>AINVNM</i> is an estimate of the 1-norm of $\text{inv}(A)$ computed in this routine.
--------------	--

Return Values

This function returns a value *info*.

= 0: Successful exit.

< 0: If *info* = -*i*, the *i*th argument had an illegal value.

?spcon

Estimates the reciprocal of the condition number of a packed symmetric matrix.

Syntax

```
lapack_int LAPACKE_sspcon( int matrix_layout, char uplo, lapack_int n, const float* ap,
const lapack_int* ipiv, float anorm, float* rcond );
```

```
lapack_int LAPACKE_dspcon( int matrix_layout, char uplo, lapack_int n, const double*
ap, const lapack_int* ipiv, double anorm, double* rcond );
```

```
lapack_int LAPACKE_cspcon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* ap, const lapack_int* ipiv, float anorm, float* rcond );
```

```
lapack_int LAPACKE_zspcon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* ap, const lapack_int* ipiv, double anorm, double* rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a packed symmetric matrix A :

$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1$ (since A is symmetric, $\kappa_\infty(A) = \kappa_1(A)$).

An estimate is obtained for $\|A^{-1}\|_1$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:

- compute $anorm$ (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?spturf](#) to compute the factorization of A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the packed upper triangular factor U of the factorization $A = U^* D U^T$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the packed lower triangular factor L of the factorization $A = L^* D L^T$.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>ap</i>	The array <i>ap</i> contains the packed factored matrix A , as returned by ?spturf . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The array <i>ipiv</i> , as returned by ?spturf .
<i>anorm</i>	The norm of the <i>original</i> matrix A (see <i>Description</i>).

Output Parameters

rcond

An estimate of the reciprocal of the condition number. The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?hpcon

Estimates the reciprocal of the condition number of a packed Hermitian matrix.

Syntax

```
lapack_int LAPACKE_chpcon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* ap, const lapack_int* ipiv, float anorm, float* rcond );

lapack_int LAPACKE_zhpcon( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* ap, const lapack_int* ipiv, double anorm, double* rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix *A*:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is Hermitian, } \kappa_\infty(A) = \kappa_1(A) \text{)}.$$

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\| \|A^{-1}\|)$.

Before calling this routine:

- compute *anorm* (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call *?hptrf* to compute the factorization of *A*.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <code>uplo = 'U'</code> , the array <i>ap</i> stores the packed upper triangular factor <i>U</i> of the factorization $A = U * D * U^T$. If <code>uplo = 'L'</code> , the array <i>ap</i> stores the packed lower triangular factor <i>L</i> of the factorization $A = L * D * L^T$.
<code>n</code>	The order of matrix <i>A</i> ; $n \geq 0$.
<code>ap</code>	The array <i>ap</i> contains the packed factored matrix <i>A</i> , as returned by ?hptrf . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$.
<code>ipiv</code>	Array, size at least $\max(1, n)$. The array <i>ipiv</i> , as returned by ?hptrf .
<code>anorm</code>	The norm of the <i>original</i> matrix <i>A</i> (see <i>Description</i>).

Output Parameters

<code>rcond</code>	An estimate of the reciprocal of the condition number. The routine sets <code>rcond = 0</code> if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <code>rcond</code> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
--------------------	--

Return Values

This function returns a value *info*.

If `info = 0`, the execution is successful.

If `info = -i`, parameter *i* had an illegal value.

Application Notes

The computed `rcond` is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $A * x = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

See Also

[Matrix Storage Schemes](#)

[?trcon](#)

Estimates the reciprocal of the condition number of a triangular matrix.

Syntax

```
lapack_int LAPACKE_strcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, const float* a, lapack_int lda, float* rcond );

lapack_int LAPACKE_dtrcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, const double* a, lapack_int lda, double* rcond );
```

```
lapack_int LAPACKE_ctrcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, const lapack_complex_float* a, lapack_int lda, float* rcond );

lapack_int LAPACKE_ztrcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, const lapack_complex_double* a, lapack_int lda, double* rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a triangular matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>norm</i>	Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', the array a stores the upper triangle of A , other array elements are not referenced. If <i>uplo</i> = 'L', the array a stores the lower triangle of A , other array elements are not referenced.
<i>diag</i>	Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array a .
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	The array a of size $\max(1, lda * n)$ contains the matrix A .
<i>lda</i>	The leading dimension of a ; $lda \geq \max(1, n)$.

Output Parameters

rcond

An estimate of the reciprocal of the condition number. The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $Ax = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

See Also

Matrix Storage Schemes

?tpcon

Estimates the reciprocal of the condition number of a packed triangular matrix.

Syntax

```
lapack_int LAPACKE_stpcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, const float* ap, float* rcond );
```

```
lapack_int LAPACKE_dtpcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, const double* ap, double* rcond );
```

```
lapack_int LAPACKE_ctpcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, const lapack_complex_float* ap, float* rcond );
```

```
lapack_int LAPACKE_ztpcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, const lapack_complex_double* ap, double* rcond );
```

Include Files

- `mk1.h`

Description

The routine estimates the reciprocal of the condition number of a packed triangular matrix *A* in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H) .$$

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>norm</i>	<p>Must be '1' or 'O' or 'I'.</p> <p>If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix <i>A</i> in 1-norm.</p> <p>If <i>norm</i> = 'I', then the routine estimates the condition number of matrix <i>A</i> in infinity-norm.</p>
<i>uplo</i>	<p>Must be 'U' or 'L'. Indicates whether <i>A</i> is upper or lower triangular:</p> <p>If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangle of <i>A</i> in packed form.</p> <p>If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangle of <i>A</i> in packed form.</p>
<i>diag</i>	<p>Must be 'N' or 'U'.</p> <p>If <i>diag</i> = 'N', then <i>A</i> is not a unit triangular matrix.</p> <p>If <i>diag</i> = 'U', then <i>A</i> is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ap</i>.</p>
<i>n</i>	The order of the matrix <i>A</i> ; $n \geq 0$.
<i>ap</i>	The array <i>ap</i> contains the packed matrix <i>A</i> . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$.

Output Parameters

<i>rcond</i>	An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
--------------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $A \cdot x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

See Also

[Matrix Storage Schemes](#)

?tbcon

Estimates the reciprocal of the condition number of a triangular band matrix.

Syntax

```
lapack_int LAPACKE_stbcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, lapack_int kd, const float* ab, lapack_int ldab, float* rcond );

lapack_int LAPACKE_dtbcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, lapack_int kd, const double* ab, lapack_int ldab, double* rcond );

lapack_int LAPACKE_ctbcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, lapack_int kd, const lapack_complex_float* ab, lapack_int ldab, float*
rcond );

lapack_int LAPACKE_ztbcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, lapack_int kd, const lapack_complex_double* ab, lapack_int ldab, double*
rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a triangular band matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H) .$$

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>norm</i>	Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangle of A in packed form. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangle of A in packed form.
<i>diag</i>	Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ab</i> .

<i>n</i>	The order of the matrix <i>A</i> ; $n \geq 0$.
<i>kd</i>	The number of superdiagonals or subdiagonals in the matrix <i>A</i> ; $kd \geq 0$.
<i>ab</i>	The array <i>ab</i> of size $\max(1, ldab*n)$ contains the band matrix <i>A</i> .
<i>ldab</i>	The leading dimension of the array <i>ab</i> . ($ldab \geq kd + 1$).

Output Parameters

<i>rcond</i>	An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
--------------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2*n(kd + 1)$ floating-point operations for real flavors and $8*n(kd + 1)$ operations for complex flavors.

See Also

Matrix Storage Schemes

Refining the Solution and Estimating Its Error: LAPACK Computational Routines

This section describes the LAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see [Routines for Matrix Factorization](#) and [Routines for Solving Systems of Linear Equations](#)).

?gerfs

Refines the solution of a system of linear equations with a general coefficient matrix and estimates its error.

Syntax

```
lapack_int LAPACKE_sgerfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf, const
lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr,
float* berr );
```

```
lapack_int LAPACKE_dgerfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf, const
lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double*
ferr, double* berr );
```

```
lapack_int LAPACKE_cgerfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af,
lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_zgerfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af,
lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int
ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^T X = B$ or $A^H X = B$ or $A^H X = B$ with a general matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?getrf](#)
- call the solver routine [?getrs](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>trans</i>	Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A^T X = B$. If <i>trans</i> = 'T', the system has the form $A^T X = B$. If <i>trans</i> = 'C', the system has the form $A^H X = B$.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>a, af, b, x</i>	Arrays: <i>a</i> (size $\max(1, lda * n)$) contains the original matrix A , as supplied to ?getrf . <i>af</i> (size $\max(1, ldaf * n)$) contains the factored matrix A , as returned by ?getrf . <i>b</i> of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout contains the right-hand side matrix B .

xof size $\max(1, \text{ldx} * \text{nrhs})$ for column major layout and $\max(1, \text{ldx} * n)$ for row major layout contains the solution matrix X .

<i>lda</i>	The leading dimension of <i>a</i> ; $\text{lda} \geq \max(1, n)$.
<i>ldaf</i>	The leading dimension of <i>af</i> ; $\text{ldaf} \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $\text{ldb} \geq \max(1, n)$ for column major layout and $\text{ldb} \geq \text{nrhs}$ for row major layout.
<i>ldx</i>	The leading dimension of <i>x</i> ; $\text{ldx} \geq \max(1, n)$ for column major layout and $\text{ldx} \geq \text{nrhs}$ for row major layout.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by <code>?getrf</code> .

Output Parameters

<i>x</i>	The refined solution matrix X .
<i>ferr, berr</i>	Arrays, size at least $\max(1, \text{nrhs})$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A * x = b$ with the same coefficient matrix A and different right hand sides b ; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

`?gerfsx`

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a general coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
lapack_int LAPACKE_sgerfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf,
const lapack_int* ipiv, const float* r, const float* c, const float* b, lapack_int ldb,
float* x, lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float*
err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_dgerfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf,
const lapack_int* ipiv, const double* r, const double* c, const double* b, lapack_int
ldb, double* x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds,
double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );

lapack_int LAPACKE_cgerfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* af, lapack_int ldaf, const lapack_int* ipiv, const float* r,
const float* c, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm,
float* err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_zgerfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* af, lapack_int ldaf, const lapack_int* ipiv, const double* r,
const double* c, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double*
x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double*
err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );
```

Include Files

- mkl.h

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed*, *r*, and *c* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

<i>matrix_layout</i>	Specifies whether two-dimensional array storage is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>trans</i>	Must be 'N', 'T', or 'C'. Specifies the form of the system of equations: If <i>trans</i> = 'N', the system has the form $A \cdot X = B$ (No transpose). If <i>trans</i> = 'T', the system has the form $A^T \cdot X = B$ (Transpose). If <i>trans</i> = 'C', the system has the form $A^H \cdot X = B$ (Conjugate transpose for complex flavors, Transpose for real flavors).

<i>equed</i>	<p>Must be 'N', 'R', 'C', or 'B'.</p> <p>Specifies the form of equilibration that was done to <i>A</i> before calling this routine.</p> <p>If <i>equed</i> = 'N', no equilibration was done.</p> <p>If <i>equed</i> = 'R', row equilibration was done, that is, <i>A</i> has been premultiplied by <i>diag(r)</i>.</p> <p>If <i>equed</i> = 'C', column equilibration was done, that is, <i>A</i> has been postmultiplied by <i>diag(c)</i>.</p> <p>If <i>equed</i> = 'B', both row and column equilibration was done, that is, <i>A</i> has been replaced by <i>diag(r)*A*diag(c)</i>. The right-hand side <i>B</i> has been changed accordingly.</p>
<i>n</i>	The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.
<i>a, af, b</i>	<p>Arrays: <i>a</i> (size $\max(1, lda*n)$), <i>af</i> (size $\max(1, ldaf*n)$), <i>b</i> (size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout).</p> <p>The array <i>a</i> contains the original <i>n</i>-by-<i>n</i> matrix <i>A</i>.</p> <p>The array <i>af</i> contains the factored form of the matrix <i>A</i>, that is, the factors <i>L</i> and <i>U</i> from the factorization $A = P*L*U$ as computed by ?getrf.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.</p>
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. Contains the pivot indices as computed by ?getrf ; for row $1 \leq i \leq n$, row <i>i</i> of the matrix was interchanged with row <i>ipiv(i)</i> .
<i>r, c</i>	<p>Arrays: <i>r</i> (size <i>n</i>), <i>c</i> (size <i>n</i>). The array <i>r</i> contains the row scale factors for <i>A</i>, and the array <i>c</i> contains the column scale factors for <i>A</i>.</p> <p><i>equed</i> = 'R' or 'B', <i>A</i> is multiplied on the left by <i>diag(r)</i>; if <i>equed</i> = 'N' or 'C', <i>r</i> is not accessed.</p> <p>If <i>equed</i> = 'R' or 'B', each element of <i>r</i> must be positive.</p> <p>If <i>equed</i> = 'C' or 'B', <i>A</i> is multiplied on the right by <i>diag(c)</i>; if <i>equed</i> = 'N' or 'R', <i>c</i> is not accessed.</p> <p>If <i>equed</i> = 'C' or 'B', each element of <i>c</i> must be positive.</p> <p>Each element of <i>r</i> or <i>c</i> should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or</p>

overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

The leading dimension of the array *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

x

Array, of size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout.

The solution matrix *X* as computed by `?getrs`

ldx

The leading dimension of the output array *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

n_err_bnds

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See `err_bnds_norm` and `err_bnds_comp` descriptions in *Output Arguments* section below.

nparams

Specifies the number of parameters set in *params*. If ≤ 0 , the *params* array is never referenced and default values are used.

params

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params[0] : Whether to perform iterative refinement or not. Default: 1.0

=0.0

No refinement is performed and no error bounds are computed.

=1.0

Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

params[1] : Maximum number of residual computations allowed for refinement.

Default

10.0

Aggressive

Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

`params[2]` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

Output Parameters

<code>x</code>	The improved solution matrix X .
<code>rcond</code>	Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If <code>rcond</code> is less than the machine precision, in particular, if <code>rcond</code> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.
<code>berr</code>	Array, size at least <code>max(1, nrhs)</code> . Contains the componentwise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of A or B that makes x_j an exact solution.
<code>err_bnds_norm</code>	Array of size <code>nrhs*n_err_bnds</code> . For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the i -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold <code>sqrt(n)*slamch(ε)</code> for single precision flavors and <code>sqrt(n)*dlamch(ε)</code> for double precision flavors.
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold <code>sqrt(n)*slamch(ε)</code> for single precision flavors and <code>sqrt(n)*dlamch(ε)</code> for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z=s*a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error `err` is stored in:

- Column major layout: `err_bnds_norm[(err - 1)*nrhs + i - 1]`.
- Row major layout: `err_bnds_norm[err - 1 + (i - 1)*n_err_bnds]`

`err_bnds_comp`

Array of size `nrhs*n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params[2] = 0.0`), then `err_bnds_comp` is not accessed. If `n_err_bnds < 3`, then at most the first `n_err_bnds` columns of the `err_bnds_comp` array are returned.

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double

precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq \text{nrhs}$, and type of error `err` is stored in:

- Column major layout: `err_bnds_comp[(err - 1) * nrhs + i - 1]`.
- Row major layout: `err_bnds_comp[err - 1 + (i - 1) * n_err_bnds]`

`params`

Output parameter only if the input contains erroneous values, namely, in `params[0]`, `params[1]`, `params[2]`. In such a case, the corresponding elements of `params` are filled with default values on output.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If `info = -i`, parameter i had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; `rcond = 0` is returned.

If `info = n+j`: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested `params[2] = 0.0`, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that for column major layout `err_bnds_norm[j - 1] = 0.0` or `err_bnds_comp[j - 1] = 0.0`; or for row major layout `err_bnds_norm[(j - 1) * n_err_bnds] = 0.0` or `err_bnds_comp[(j - 1) * n_err_bnds] = 0.0`). See the definition of `err_bnds_norm` and `err_bnds_comp` for `err = 1`. To get information about all of the right-hand sides, check `err_bnds_norm` or `err_bnds_comp`.

See Also

[Matrix Storage Schemes](#)

?gbrfs

Refines the solution of a system of linear equations with a general band coefficient matrix and estimates its error.

Syntax

```
lapack_int LAPACKE_sgbrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl,
lapack_int ku, lapack_int nrhs, const float* ab, lapack_int ldab, const float* afb,
lapack_int ldafb, const lapack_int* ipiv, const float* b, lapack_int ldb, float* x,
lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dgbrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl,
lapack_int ku, lapack_int nrhs, const double* ab, lapack_int ldab, const double* afb,
lapack_int ldafb, const lapack_int* ipiv, const double* b, lapack_int ldb, double* x,
lapack_int ldx, double* ferr, double* berr );
```

```
lapack_int LAPACKE_cgbrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl,
lapack_int ku, lapack_int nrhs, const lapack_complex_float* ab, lapack_int ldab, const
lapack_complex_float* afb, lapack_int ldafb, const lapack_int* ipiv, const
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* ferr, float* berr );
```

```
lapack_int LAPACKE_zgbrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl,
lapack_int ku, lapack_int nrhs, const lapack_complex_double* ab, lapack_int ldab, const
lapack_complex_double* afb, lapack_int ldafb, const lapack_int* ipiv, const
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ or $A^T X = B$ or $A^H X = B$ with a band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?gbtrf](#)
- call the solver routine [?gbtrs](#).

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

trans Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If $trans = 'N'$, the system has the form $A * X = B$.

If $trans = 'T'$, the system has the form $A^T * X = B$.

If $trans = 'C'$, the system has the form $A^H * X = B$.

n The order of the matrix A ; $n \geq 0$.

kl The number of sub-diagonals within the band of A ; $kl \geq 0$.

ku The number of super-diagonals within the band of A ; $ku \geq 0$.

$nrhs$ The number of right-hand sides; $nrhs \geq 0$.

ab,afb,b,x

Arrays:

ab (size $\max(1, ldab * n)$) contains the original band matrix A , as supplied to `?gbtrf`, but stored in rows from 1 to $kl + ku + 1$ for column major layout, and columns from 1 to $kl + ku + 1$ for row major layout.

afb (size $\max(1, ldafb * n)$) contains the factored band matrix A , as returned by `?gbtrf`.

b of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout contains the right-hand side matrix B .

x of size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout contains the solution matrix X .

$ldab$ The leading dimension of ab , $ldab \geq kl + ku + 1$.

$ldaafb$ The leading dimension of afb , $ldaafb \geq 2 * kl + ku + 1$.

ldb The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ldx The leading dimension of x ; $ldx \geq \max(1, n)$.

$ipiv$ Array, size at least $\max(1, n)$. The $ipiv$ array, as returned by `?gbtrf`.

Output Parameters

x The refined solution matrix X .

$ferr, berr$ Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.

If $info = 0$, the execution is successful.

If $info = -i$, parameter i had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n(kl + ku)$ floating-point operations (for real flavors) or $16n(kl + ku)$ operations (for complex flavors). In addition, each step of iterative refinement involves $2n(4kl + 3ku)$ operations (for real flavors) or $8n(4kl + 3ku)$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?gbrfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a banded coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
lapack_int LAPACKE_sgbrfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, const float* ab, lapack_int ldab, const
float* afb, lapack_int ldafb, const lapack_int* ipiv, const float* r, const float* c,
const float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* berr,
lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams,
float* params );
```

```
lapack_int LAPACKE_dgbrfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, const double* ab, lapack_int ldab, const
double* afb, lapack_int ldafb, const lapack_int* ipiv, const double* r, const double*
c, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double*
berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int
nparams, double* params );
```

```
lapack_int LAPACKE_cgbrfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, const lapack_complex_float* ab,
lapack_int ldab, const lapack_complex_float* afb, lapack_int ldafb, const lapack_int*
ipiv, const float* r, const float* c, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* rcond, float* berr, lapack_int
n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float*
params );
```

```
lapack_int LAPACKE_zgbrfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, const lapack_complex_double* ab,
lapack_int ldab, const lapack_complex_double* afb, lapack_int ldafb, const lapack_int*
ipiv, const double* r, const double* c, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* berr, lapack_int
n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double*
params );
```

Include Files

- mkl.h

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed*, *r*, and *c* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

<i>matrix_layout</i>	Specifies whether two-dimensional array storage is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>trans</i>	Must be 'N', 'T', or 'C'. Specifies the form of the system of equations: If <i>trans</i> = 'N', the system has the form $A * X = B$ (No transpose). If <i>trans</i> = 'T', the system has the form $A^T * X = B$ (Transpose). If <i>trans</i> = 'C', the system has the form $A^H * X = B$ (Conjugate transpose for complex flavors, Transpose for real flavors).
<i>equed</i>	Must be 'N', 'R', 'C', or 'B'. Specifies the form of equilibration that was done to <i>A</i> before calling this routine. If <i>equed</i> = 'N', no equilibration was done. If <i>equed</i> = 'R', row equilibration was done, that is, <i>A</i> has been premultiplied by <i>diag(r)</i> . If <i>equed</i> = 'C', column equilibration was done, that is, <i>A</i> has been postmultiplied by <i>diag(c)</i> . If <i>equed</i> = 'B', both row and column equilibration was done, that is, <i>A</i> has been replaced by $diag(r) * A * diag(c)$. The right-hand side <i>B</i> has been changed accordingly.
<i>n</i>	The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	The number of subdiagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>nrhs</i>	The number of right-hand sides; the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.
<i>ab, afb, b</i>	The array <i>abof</i> size $\max(1, ldab * n)$ contains the original matrix <i>A</i> in band storage, in rows from 1 to $kl + ku + 1$ for column major layout, and in columns from 1 to $kl + ku + 1$ for row major layout. The array <i>afb</i> size $\max(1, ldafb * n)$ contains details of the LU factorization of the banded matrix <i>A</i> as computed by ?gbtrf .

	The array <i>bof</i> size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
<i>ldab</i>	The leading dimension of the array <i>ab</i> ; $ldab \geq kl + ku + 1$.
<i>ldaafb</i>	The leading dimension of the array <i>afb</i> ; $ldaafb \geq 2*kl + ku + 1$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. Contains the pivot indices as computed by ?gbtrf ; for row $1 \leq i \leq n$, row <i>i</i> of the matrix was interchanged with row <i>ipiv</i> [<i>i</i> -1].
<i>r, c</i>	<p>Arrays: <i>r</i>(<i>n</i>), <i>c</i>(<i>n</i>). The array <i>r</i> contains the row scale factors for <i>A</i>, and the array <i>c</i> contains the column scale factors for <i>A</i>.</p> <p>If <i>equed</i> = 'R' or 'B', <i>A</i> is multiplied on the left by <i>diag</i>(<i>r</i>); if <i>equed</i> = 'N' or 'C', <i>r</i> is not accessed.</p> <p>If <i>equed</i> = 'R' or 'B', each element of <i>r</i> must be positive.</p> <p>If <i>equed</i> = 'C' or 'B', <i>A</i> is multiplied on the right by <i>diag</i>(<i>c</i>); if <i>equed</i> = 'N' or 'R', <i>c</i> is not accessed.</p> <p>If <i>equed</i> = 'C' or 'B', each element of <i>c</i> must be positive.</p> <p>Each element of <i>r</i> or <i>c</i> should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.</p>
<i>ldb</i>	The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>x</i>	<p>Array, size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout.</p> <p>The solution matrix <i>X</i> as computed by sgbtrs/dgbtrs for real flavors or cgbtrs/zgbtrs for complex flavors.</p>
<i>ldx</i>	The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.
<i>n_err_bnds</i>	Number of error bounds to return for each right-hand side and each type (normwise or componentwise). See err_bnds_norm and err_bnds_comp descriptions in <i>Output Arguments</i> section below.
<i>nparams</i>	Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.
<i>params</i>	<p>Array, size <i>nparams</i>. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument.</p> <p><i>params</i>[0] : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).</p>

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

`params[1]` : Maximum number of residual computations allowed for refinement.

Default	10.0
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy.

`params[2]` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

Output Parameters

<code>x</code>	The improved solution matrix <i>X</i> .
<code>rcond</code>	Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <code>rcond</code> is less than the machine precision, in particular, if <code>rcond</code> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.
<code>berr</code>	Array, size at least <code>max(1, nrhs)</code> . Contains the componentwise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes x_j an exact solution.
<code>err_bnds_norm</code>	Array of size <code>nrhs*n_err_bnds</code> . For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows: Normwise relative error in the <i>i</i> -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z=s*a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error `err` is stored in:

- Column major layout: `err_bnds_norm[(err - 1)*nrhs + i - 1]`.
- Row major layout: `err_bnds_norm[err - 1 + (i - 1)*n_err_bnds]`

`err_bnds_comp`

Array of size `nrhs*n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params[2] = 0.0`), then `err_bnds_comp` is not accessed.

`err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3` Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq \text{nrhs}$, and type of error `err` is stored in:

- Column major layout: `err_bnds_comp[(err - 1) * nrhs + i - 1]`.
- Row major layout: `err_bnds_comp[err - 1 + (i - 1) * n_err_bnds]`

`params`

Output parameter only if the input contains erroneous values, namely, in `params[0]`, `params[1]`, and `params[2]`. In such a case, the corresponding elements of `params` are filled with default values on output.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If `info = -i`, parameter i had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; `rcond = 0` is returned.

If `info = n+j`: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested `params[2] = 0.0`, then the j -th

right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that for column major layout $err_bnds_norm[j - 1] = 0.0$ or $err_bnds_comp[j - 1] = 0.0$; or for row major layout $err_bnds_norm[(j - 1)*n_err_bnds] = 0.0$ or $err_bnds_comp[(j - 1)*n_err_bnds] = 0.0$). See the definition of err_bnds_norm and err_bnds_comp for $err = 1$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

See Also

Matrix Storage Schemes

?gtrfs

Refines the solution of a system of linear equations with a tridiagonal coefficient matrix and estimates its error.

Syntax

```
lapack_int LAPACKE_sgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const float* dl, const float* d, const float* du, const float* dlf, const float*
df, const float* duf, const float* du2, const lapack_int* ipiv, const float* b,
lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_dgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const double* dl, const double* d, const double* du, const double* dlf, const
double* df, const double* duf, const double* du2, const lapack_int* ipiv, const double*
b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );

lapack_int LAPACKE_cgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const lapack_complex_float* dl, const lapack_complex_float* d, const
lapack_complex_float* du, const lapack_complex_float* dlf, const lapack_complex_float*
df, const lapack_complex_float* duf, const lapack_complex_float* du2, const lapack_int*
ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_zgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const lapack_complex_double* dl, const lapack_complex_double* d, const
lapack_complex_double* du, const lapack_complex_double* dlf, const
lapack_complex_double* df, const lapack_complex_double* duf, const
lapack_complex_double* du2, const lapack_int* ipiv, const lapack_complex_double* b,
lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a tridiagonal matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}|/|a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i|/|b_i| \leq \beta |b_i| \quad \text{such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?gttrf](#)

- call the solver routine [?gttrs](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>trans</i>	Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A * X = B$. If <i>trans</i> = 'T', the system has the form $A^T * X = B$. If <i>trans</i> = 'C', the system has the form $A^H * X = B$.
<i>n</i>	The order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, that is, the number of columns of the matrix <i>B</i> ; $nrhs \geq 0$.
<i>dl</i>	Array <i>dl</i> of size $n - 1$ contains the subdiagonal elements of <i>A</i> .
<i>d</i>	Array <i>d</i> of size n contains the diagonal elements of <i>A</i> .
<i>du</i>	Array <i>du</i> of size $n - 1$ contains the superdiagonal elements of <i>A</i> .
<i>dlf</i>	Array <i>dlf</i> of size $n - 1$ contains the $(n - 1)$ multipliers that define the matrix <i>L</i> from the <i>LU</i> factorization of <i>A</i> as computed by ?gttrf .
<i>df</i>	Array <i>df</i> of size n contains the n diagonal elements of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i> .
<i>duf</i>	Array <i>duf</i> of size $n - 1$ contains the $(n - 1)$ elements of the first superdiagonal of <i>U</i> .
<i>du2</i>	Array <i>du2</i> of size $n - 2$ contains the $(n - 2)$ elements of the second superdiagonal of <i>U</i> .
<i>b</i>	Array <i>b</i> (size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout) contains the right-hand side matrix <i>B</i> .
<i>x</i>	Array <i>x</i> (size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout) contains the solution matrix <i>X</i> , as computed by ?gttrs .
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ldx</i>	The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?gttrf .

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

See Also

Matrix Storage Schemes

?porfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite coefficient matrix and estimates its error.

Syntax

```
lapack_int LAPACKE_sporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* a, lapack_int lda, const float* af, lapack_int ldaf, const float* b,
lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* a, lapack_int lda, const double* af, lapack_int ldaf, const double* b,
lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );
```

```
lapack_int LAPACKE_cporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af,
lapack_int ldaf, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float*
x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_zporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af,
lapack_int ldaf, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double*
x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a symmetric (Hermitian) positive definite matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?potrf](#)
- call the solver routine [?potrs](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	The order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>a</i>	Array <i>a</i> (size $\max(1, lda*n)$) contains the original matrix <i>A</i> , as supplied to ?potrf .
<i>af</i>	Array <i>af</i> (size $\max(1, ldaf*n)$) contains the factored matrix <i>A</i> , as returned by ?potrf .
<i>b</i>	Array <i>b</i> of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the right-hand side matrix <i>B</i> . The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>x</i>	Array <i>x</i> of size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout contains the solution matrix <i>X</i> .
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ldx</i>	The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?porfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric/Hermitian positive-definite coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
lapack_int LAPACKE_sporfsx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf,
const float* s, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond,
float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp,
lapack_int nparams, float* params );
```

```
lapack_int LAPACKE_dporfsx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf,
const double* s, const double* b, lapack_int ldb, double* x, lapack_int ldx, double*
rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double*
err_bnds_comp, lapack_int nparams, double* params );
```

```
lapack_int LAPACKE_cporfsx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* af, lapack_int ldaf, const float* s, const lapack_complex_float*
b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* berr,
lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams,
float* params );
```

```
lapack_int LAPACKE_zporfsx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* af, lapack_int ldaf, const double* s, const
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double*
err_bnds_comp, lapack_int nparams, double* params );
```

Include Files

- mkl.h

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters `equed` and `s` below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

<i>matrix_layout</i>	Specifies whether two-dimensional array storage is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored:</p> <p>If <i>uplo</i> = 'U', the upper triangle of A is stored.</p> <p>If <i>uplo</i> = 'L', the lower triangle of A is stored.</p>
<i>equed</i>	<p>Must be 'N' or 'Y'.</p> <p>Specifies the form of equilibration that was done to A before calling this routine.</p> <p>If <i>equed</i> = 'N', no equilibration was done.</p> <p>If <i>equed</i> = 'Y', both row and column equilibration was done, that is, A has been replaced by $diag(s) * A * diag(s)$. The right-hand side B has been changed accordingly.</p>
<i>n</i>	The number of linear equations; the order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<i>a</i>	The array <i>a</i> (size $\max(1, lda * n)$) contains the symmetric/Hermitian matrix A as specified by <i>uplo</i> . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of <i>a</i> contains the upper triangular part of the matrix A and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of <i>a</i> contains the lower triangular part of the matrix A and the strictly upper triangular part of <i>a</i> is not referenced.
<i>af</i>	The array <i>af</i> (size $\max(1, ldaf * n)$) contains the triangular factor L or U from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ as computed by spotrf for real flavors or dpotrf for complex flavors.
<i>b</i>	The array <i>b</i> (size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout) contains the matrix B whose columns are the right-hand sides for the systems of equations.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>s</i>	<p>Array of size n. The array <i>s</i> contains the scale factors for A.</p> <p>If <i>equed</i> = 'N', <i>s</i> is not accessed.</p> <p>If <i>equed</i> = 'Y', each element of <i>s</i> must be positive.</p> <p>Each element of <i>s</i> should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.</p>

<i>ldb</i>	The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.								
<i>x</i>	<p>Array, size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout.</p> <p>The solution matrix <i>X</i> as computed by <code>?potrs</code></p>								
<i>ldx</i>	The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.								
<i>n_err_bnds</i>	Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <i>err_bnds_norm</i> and <i>err_bnds_comp</i> descriptions in <i>Output Arguments</i> section below.								
<i>nparams</i>	Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.								
<i>params</i>	<p>Array, size <i>nparams</i>. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument.</p> <p><i>params</i>[0] : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).</p> <table> <tr> <td>=0.0</td><td>No refinement is performed and no error bounds are computed.</td></tr> <tr> <td>=1.0</td><td>Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.</td></tr> </table> <p>(Other values are reserved for future use.)</p> <p><i>params</i>[1] : Maximum number of residual computations allowed for refinement.</p> <table> <tr> <td>Default</td><td>10.0</td></tr> <tr> <td>Aggressive</td><td>Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i>. If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.</td></tr> </table> <p><i>params</i>[2] : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).</p>	=0.0	No refinement is performed and no error bounds are computed.	=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.	Default	10.0	Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.
=0.0	No refinement is performed and no error bounds are computed.								
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.								
Default	10.0								
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.								

Output Parameters

<i>x</i>	The improved solution matrix <i>X</i> .
----------	---

rcond Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of *A* or *B* that makes x_j an exact solution.

err_bnds_norm Array of size *nrhs***n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

err=1 "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

err=2 "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3 Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error err is stored in:

- Column major layout: $err_bnds_norm[(err - 1) * nrhs + i - 1]$.
- Row major layout: $err_bnds_norm[err - 1 + (i - 1) * n_err_bnds]$

`err_bnds_comp`

Array of size $nrhs * n_err_bnds$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($params[2] = 0.0$), then `err_bnds_comp` is not accessed.

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error err is stored in:

- Column major layout: `err_bnds_comp[(err - 1)*nrhs + i - 1]`.
- Row major layout: `err_bnds_comp[err - 1 + (i - 1)*n_err_bnds]`

`params`

Output parameter only if the input contains erroneous values, namely in `params[0]`, `params[1]`, or `params[2]`. In such a case, the corresponding elements of `params` are filled with default values on output.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If `info = -i`, parameter `i` had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; `rcond = 0` is returned.

If `info = n+j`: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested `params[2] = 0.0`, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that for column major layout `err_bnds_norm[j - 1] = 0.0` or `err_bnds_comp[j - 1] = 0.0`; or for row major layout `err_bnds_norm[(j - 1)*n_err_bnds] = 0.0` or `err_bnds_comp[(j - 1)*n_err_bnds] = 0.0`). See the definition of `err_bnds_norm` and `err_bnds_comp` for `err = 1`. To get information about all of the right-hand sides, check `err_bnds_norm` or `err_bnds_comp`.

See Also

Matrix Storage Schemes

`?pprfs`

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite coefficient matrix stored in a packed format and estimates its error.

Syntax

```
lapack_int LAPACKE_spprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* ap, const float* afp, const float* b, lapack_int ldb, float* x, lapack_int
ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dpprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* ap, const double* afp, const double* b, lapack_int ldb, double* x,
lapack_int ldx, double* ferr, double* berr );
```

```
lapack_int LAPACKE_cpprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* ap, const lapack_complex_float* afp, const
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* ferr, float* berr );
```

```
lapack_int LAPACKE_zpprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* ap, const lapack_complex_double* afp, const
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* ferr, double* berr );
```

Include Files

- `mkl.h`

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ with a symmetric (Hermitian) positive definite matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution

$$\|x - x_e\|_\infty / \|x\|_\infty$$

where x_e is the exact solution.

Before calling this routine:

- call the factorization routine [?pptrf](#)
- call the solver routine [?pptrs](#).

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the upper triangle of A is stored. If <code>uplo = 'L'</code> , the lower triangle of A is stored.
<code>n</code>	The order of the matrix A ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides; $nrhs \geq 0$.
<code>ap</code>	<code>ap</code> contains the original matrix A in a packed format, as supplied to ?pptrf . The dimension of <code>ap</code> must be at least $\max(1, n(n+1)/2)$.
<code>afp</code>	<code>afp</code> contains the factored matrix A in a packed format, as returned by ?pptrf . The dimension of <code>afp</code> must be at least $\max(1, n(n+1)/2)$.
<code>b</code>	Array b of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout contains the right-hand side matrix B .
<code>x</code>	Array x of size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout contains the solution matrix X .
<code>ldb</code>	The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<code>ldx</code>	The leading dimension of x ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

x	The refined solution matrix X .
$ferr, berr$	Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A \cdot x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?pbrfs

Refines the solution of a system of linear equations with a band symmetric (Hermitian) positive-definite coefficient matrix and estimates its error.

Syntax

```
lapack_int LAPACKE_spbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
lapack_int nrhs, const float* ab, lapack_int ldab, const float* afb, lapack_int ldafb,
const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_dpbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
lapack_int nrhs, const double* ab, lapack_int ldab, const double* afb, lapack_int
ldafb, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double*
berr );

lapack_int LAPACKE_cpbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
lapack_int nrhs, const lapack_complex_float* ab, lapack_int ldab, const
lapack_complex_float* afb, lapack_int ldafb, const lapack_complex_float* b, lapack_int
ldb, lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_zpbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
lapack_int nrhs, const lapack_complex_double* ab, lapack_int ldab, const
lapack_complex_double* afb, lapack_int ldafb, const lapack_complex_double* b,
lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- `mk1.h`

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ with a symmetric (Hermitian) positive definite band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?pbtrf`
- call the solver routine `?pbtrs`.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo</code> = 'U', the upper triangle of A is stored. If <code>uplo</code> = 'L', the lower triangle of A is stored.
<code>n</code>	The order of the matrix A ; $n \geq 0$.
<code>kd</code>	The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<code>nrhs</code>	The number of right-hand sides; $nrhs \geq 0$.
<code>ab</code>	Array <code>ab</code> (size $\max(ldab, n)$) contains the original band matrix A , as supplied to <code>?pbtrf</code> .
<code>afb</code>	Array <code>afb</code> (size $\max(ldafb, n)$) contains the factored band matrix A , as returned by <code>?pbtrf</code> .
<code>b</code>	Array <code>b</code> of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout contains the right-hand side matrix B .
<code>x</code>	Array <code>x</code> of size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout contains the solution matrix X .
<code>ldab</code>	The leading dimension of <code>ab</code> ; $ldab \geq kd + 1$.
<code>ldafb</code>	The leading dimension of <code>afb</code> ; $ldafb \geq kd + 1$.
<code>ldb</code>	The leading dimension of <code>b</code> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ldx The leading dimension of *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

x The refined solution matrix *X*.

ferr, berr Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $8n \cdot kd$ floating-point operations (for real flavors) or $32n \cdot kd$ operations (for complex flavors). In addition, each step of iterative refinement involves $12n \cdot kd$ operations (for real flavors) or $48n \cdot kd$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A \cdot x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4n \cdot kd$ floating-point operations for real flavors or $16n \cdot kd$ for complex flavors.

See Also

Matrix Storage Schemes

?ptrfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal coefficient matrix and estimates its error.

Syntax

```
lapack_int LAPACKESptrfs( int matrix_layout, lapack_int n, lapack_int nrhs, const
float* d, const float* e, const float* df, const float* ef, const float* b, lapack_int
ldb, float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKEDptrfs( int matrix_layout, lapack_int n, lapack_int nrhs, const
double* d, const double* e, const double* df, const double* ef, const double* b,
lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );
```

```
lapack_int LAPACKECptrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* d, const lapack_complex_float* e, const float* df, const
lapack_complex_float* ef, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKEZptrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* d, const lapack_complex_double* e, const double* df, const
lapack_complex_double* ef, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- `mkl.h`

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ with a symmetric (Hermitian) positive definite tridiagonal matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine `?pttrf`
- call the solver routine `?pttrs`.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Used for complex flavors only. Must be 'U' or 'L'. Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and how A is factored: If <code>uplo = 'U'</code> , the array <code>e</code> stores the superdiagonal of A , and A is factored as $U^H * D * U$. If <code>uplo = 'L'</code> , the array <code>e</code> stores the subdiagonal of A , and A is factored as $L * D * L^H$.
<code>n</code>	The order of the matrix A ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides; $nrhs \geq 0$.
<code>d</code>	The array <code>d</code> (size n) contains the n diagonal elements of the tridiagonal matrix A .
<code>df</code>	The array <code>df</code> (size n) contains the n diagonal elements of the diagonal matrix D from the factorization of A as computed by <code>?pttrf</code> .
<code>e, ef, b, x</code>	The array <code>e</code> (size $n - 1$) contains the $(n - 1)$ off-diagonal elements of the tridiagonal matrix A (see <code>uplo</code>). The array <code>ef</code> (size $n - 1$) contains the $(n - 1)$ off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by <code>?pttrf</code> (see <code>uplo</code>). The array <code>b</code> of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout contains the matrix B whose columns are the right-hand sides for the systems of equations.

The array `x` of size $\max(1, \text{ldx} * \text{nrhs})$ for column major layout and $\max(1, \text{ldx} * n)$ for row major layout contains the solution matrix `X` as computed by `?pttrs`.

`ldb`

The leading dimension of `b`; $\text{ldb} \geq \max(1, n)$ for column major layout and $\text{ldb} \geq \text{nrhs}$ for row major layout.

`ldx`

The leading dimension of `x`; $\text{ldx} \geq \max(1, n)$ for column major layout and $\text{ldx} \geq \text{nrhs}$ for row major layout.

Output Parameters

`x`

The refined solution matrix `X`.

`ferr, berr`

Arrays, size at least $\max(1, \text{nrhs})$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, parameter `i` had an illegal value.

See Also

Matrix Storage Schemes

`?syrfs`

Refines the solution of a system of linear equations with a symmetric coefficient matrix and estimates its error.

Syntax

```
lapack_int LAPACKE_ssyrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* a, lapack_int lda, const float* af, lapack_int ldaf, const lapack_int*
ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float*
berr );
```

```
lapack_int LAPACKE_dsyrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* a, lapack_int lda, const double* af, lapack_int ldaf, const lapack_int*
ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double*
berr );
```

```
lapack_int LAPACKE_csyrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af,
lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_zsyrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af,
lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int
ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- `mk1.h`

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A * X = B$ with a symmetric full-storage matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?sytrf](#)
- call the solver routine [?sytrs](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>a</i>	Array <i>a</i> (size $\max(1, lda * n)$) contains the original matrix A , as supplied to ?sytrf .
<i>af</i>	Array <i>af</i> (size $\max(1, ldaf * n)$) contains the factored matrix A , as returned by ?sytrf .
<i>b</i>	Array <i>b</i> of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout contains the right-hand side matrix B .
<i>x</i>	Array <i>x</i> of size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout contains the solution matrix X .
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ldx</i>	The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sytrf .

Output Parameters

<code>x</code>	The refined solution matrix X .
<code>ferr, berr</code>	Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, parameter `i` had an illegal value.

Application Notes

The bounds returned in `ferr` are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A \cdot x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?syrfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric indefinite coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
lapack_int LAPACK_essyrfsx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf,
const lapack_int* ipiv, const float* s, const float* b, lapack_int ldb, float* x,
lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm,
float* err_bnds_comp, lapack_int nparams, float* params );
```

```
lapack_int LAPACK_dsyrfx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf,
const lapack_int* ipiv, const double* s, const double* b, lapack_int ldb, double* x,
lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double*
err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );
```

```
lapack_int LAPACK_csyrfx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* af, lapack_int ldaf, const lapack_int* ipiv, const float* s,
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float*
err_bnds_comp, lapack_int nparams, float* params );
```

```
lapack_int LAPACK_zsyrfx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* af, lapack_int ldaf, const lapack_int* ipiv, const double* s,
```

```
const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int
ldx, double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double*
err_bnds_comp, lapack_int nparams, double* params );
```

Include Files

- mkl.h

Description

The routine improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed* and *s* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

<i>matrix_layout</i>	Specifies whether two-dimensional array storage is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is stored: If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>equed</i>	Must be 'N' or 'Y'. Specifies the form of equilibration that was done to <i>A</i> before calling this routine. If <i>equed</i> = 'N', no equilibration was done. If <i>equed</i> = 'Y', both row and column equilibration was done, that is, <i>A</i> has been replaced by $diag(s) * A * diag(s)$. The right-hand side <i>B</i> has been changed accordingly.
<i>n</i>	The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.
<i>a</i> , <i>af</i> , <i>b</i>	The array <i>a</i> (size $\max(1, lda * n)$) contains the symmetric/Hermitian matrix <i>A</i> as specified by <i>uplo</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> and the strictly upper triangular part of <i>a</i> is not referenced. The array <i>af</i> (size $\max(1, ldaf * n)$) contains the triangular factor <i>L</i> or <i>U</i> from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ as computed by ssytrf for real flavors or dsytrf for complex flavors.

The array *b* (size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout) contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.				
<i>ldaf</i>	The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.				
<i>ipiv</i>	Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i> as determined by ssytrf for real flavors or dsytrf for complex flavors.				
<i>s</i>	<p>Array, size (<i>n</i>). The array <i>s</i> contains the scale factors for <i>A</i>.</p> <p>If <i>equed</i> = 'N', <i>s</i> is not accessed.</p> <p>If <i>equed</i> = 'Y', each element of <i>s</i> must be positive.</p> <p>Each element of <i>s</i> should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.</p>				
<i>ldb</i>	The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.				
<i>x</i>	<p>Array, of size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout.</p> <p>The solution matrix <i>X</i> as computed by ?sytrs</p>				
<i>ldx</i>	The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.				
<i>n_err_bnds</i>	Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <i>err_bnds_norm</i> and <i>err_bnds_comp</i> descriptions in <i>Output Arguments</i> section below.				
<i>nparams</i>	Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.				
<i>params</i>	<p>Array, size <i>nparams</i>. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument.</p> <p><i>params</i>[0] : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).</p> <table> <tr> <td>=0.0</td><td>No refinement is performed and no error bounds are computed.</td></tr> <tr> <td>=1.0</td><td>Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.</td></tr> </table>	=0.0	No refinement is performed and no error bounds are computed.	=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.
=0.0	No refinement is performed and no error bounds are computed.				
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.				

(Other values are reserved for future use.)

`params[1]` : Maximum number of residual computations allowed for refinement.

Default 10.0

Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

`params[2]` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

Output Parameters

`x` The improved solution matrix *X*.

`rcond` Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If `rcond` is less than the machine precision, in particular, if `rcond` = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

`berr` Array, size at least `max(1, nrhs)`. Contains the componentwise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of *A* or *B* that makes x_j an exact solution.

`err_bnds_norm` Array of size `nrhs*n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:
Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

`err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold `sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors.

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is

greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error `err` is stored in:

- Column major layout: `err_bnds_norm[(err - 1)*nrhs + i - 1]`.
- Row major layout: `err_bnds_norm[err - 1 + (i - 1)*n_err_bnds]`

`err_bnds_comp`

Array of size $nrhs * n_err_bnds$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params[2] = 0.0`), then `err_bnds_comp` is not accessed.

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq \text{nrhs}$, and type of error `err` is stored in:

- Column major layout: `err_bnds_comp[(err - 1)*nrhs + i - 1]`.
- Row major layout: `err_bnds_comp[err - 1 + (i - 1)*n_err_bnds]`

`params`

Output parameter only if the input contains erroneous values, namely, in `params[0]`, `params[1]`, `params[2]`. In such a case, the corresponding elements of `params` are filled with default values on output.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If `info = -i`, parameter i had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; `rcond = 0` is returned.

If `info = n+j`: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested `params[2] = 0.0`, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that for column major layout `err_bnds_norm[j - 1] = 0.0` or `err_bnds_comp[j - 1] = 0.0`; or for row major layout `err_bnds_norm[(j - 1)*n_err_bnds] = 0.0` or `err_bnds_comp[(j - 1)*n_err_bnds] = 0.0`). See the definition of `err_bnds_norm` and `err_bnds_comp` for `err = 1`. To get information about all of the right-hand sides, check `err_bnds_norm` or `err_bnds_comp`.

See Also

Matrix Storage Schemes

`?herfs`

Refines the solution of a system of linear equations with a complex Hermitian coefficient matrix and estimates its error.

Syntax

```
lapack_int LAPACKE_cherfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af,
lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_zherfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af,
lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int
ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a complex Hermitian full-storage matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error*. This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?hetrf](#)
- call the solver routine [?hetrs](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>a, af, b, x</i>	Arrays: <i>a</i> (size $\max(1, lda \cdot n)$) contains the original matrix A , as supplied to ?hetrf . <i>af</i> (size $\max(1, ldaf \cdot n)$) contains the factored matrix A , as returned by ?hetrf . <i>b</i> of size $\max(1, ldb \cdot nrhs)$ for column major layout and $\max(1, ldb \cdot n)$ for row major layout contains the right-hand side matrix B .

x of size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout contains the solution matrix X .

lda

The leading dimension of a ; $lda \geq \max(1, n)$.

ldaf

The leading dimension of af ; $ldaf \geq \max(1, n)$.

ldb

The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ldx

The leading dimension of x ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

ipiv

Array, size at least $\max(1, n)$. The *ipiv* array, as returned by [?hetrf](#).

Output Parameters

x

The refined solution matrix X .

ferr, berr

Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is [?ssyrfs/?dsyrfs](#)

See Also

Matrix Storage Schemes

[?herfsx](#)

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric indefinite coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```
lapack_int LAPACKE_cherfsx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* af, lapack_int ldaf, const lapack_int* ipiv, const float* s,
```



```

const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float*
err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_zherfsx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* af, lapack_int ldaf, const lapack_int* ipiv, const double* s,
const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int
ldx, double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double*
err_bnds_comp, lapack_int nparams, double* params );

```

Include Files

- mkl.h

Description

The routine improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed* and *s* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

<i>matrix_layout</i>	Specifies whether two-dimensional array storage is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is stored: If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>equed</i>	Must be 'N' or 'Y'. Specifies the form of equilibration that was done to <i>A</i> before calling this routine. If <i>equed</i> = 'N', no equilibration was done. If <i>equed</i> = 'Y', both row and column equilibration was done, that is, <i>A</i> has been replaced by <i>diag(s)*A*diag(s)</i> . The right-hand side <i>B</i> has been changed accordingly.
<i>n</i>	The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.
<i>a</i> , <i>af</i> , <i>b</i>	The array <i>a</i> of size $\max(1, lda*n)$ contains the Hermitian matrix <i>A</i> as specified by <i>uplo</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> and the strictly lower

triangular part of a is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A and the strictly upper triangular part of a is not referenced.

The array af of size $\max(1, ldaf*n)$ contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by [ssytrf](#) for `cherfsx` or [dsytrf](#) for `zherfsx`.

The array b of size $\max(1, ldb*nrhs)$ for row major layout and $\max(1, ldb*n)$ for column major layout contains the matrix B whose columns are the right-hand sides for the systems of equations.

<code>lda</code>	The leading dimension of a ; $lda \geq \max(1, n)$.
<code>ldaf</code>	The leading dimension of af ; $ldaf \geq \max(1, n)$.
<code>ipiv</code>	Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D as determined by ssytrf for real flavors or dsytrf for complex flavors.
<code>s</code>	<p>Array, size (n). The array s contains the scale factors for A.</p> <p>If $equed = 'N'$, s is not accessed.</p> <p>If $equed = 'Y'$, each element of s must be positive.</p> <p>Each element of s should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.</p>
<code>ldb</code>	The leading dimension of the array b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<code>x</code>	<p>Array, size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout.</p> <p>The solution matrix X as computed by ?hetrs</p>
<code>ldx</code>	The leading dimension of the output array x ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.
<code>n_err_bnds</code>	Number of error bounds to return for each right hand side and each type (normwise or componentwise). See err_bnds_norm and err_bnds_comp descriptions in <i>Output Arguments</i> section below.
<code>nparams</code>	Specifies the number of parameters set in <code>params</code> . If ≤ 0 , the <code>params</code> array is never referenced and default values are used.
<code>params</code>	<p>Array, size <code>nparams</code>. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to <code>nparams</code> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <code>nparams = 0</code>, which prevents the source code from accessing the <code>params</code> argument.</p> <p><code>params[0]</code> : Whether to perform iterative refinement or not. Default: 1.0 (for <code>cherfsx</code>), 1.0D+0 (for <code>zherfsx</code>).</p>

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

`params[1]` : Maximum number of residual computations allowed for refinement.

Default	10
Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy.

`params[2]` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

Output Parameters

<code>x</code>	The improved solution matrix <i>X</i> .
<code>rcond</code>	Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <code>rcond</code> is less than the machine precision, in particular, if <code>rcond</code> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.
<code>berr</code>	Array, size at least <code>max(1, nrhs)</code> . Contains the componentwise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes x_j an exact solution.
<code>err_bnds_norm</code>	Array of size <code>nrhs*n_err_bnds</code> . For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows: Normwise relative error in the <i>i</i> -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>cherfsx</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>zherfsx</code> .
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>cherfsx</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>zherfsx</code> . This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error `err` is stored in:

- Column major layout: `err_bnds_norm[(err - 1)*nrhs + i - 1]`.
- Row major layout: `err_bnds_norm[err - 1 + (i - 1)*n_err_bnds]`

`err_bnds_comp`

Array of size `nrhs*n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params[2] = 0.0`), then `err_bnds_comp` is not accessed.

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>cherfsx</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>zherfsx</code> .
--------------------	---

err=2 "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for *cherfsx* and $\sqrt{n} * \text{dlamch}(\epsilon)$ for *zherfsx*. This error bound should only be trusted if the previous boolean is true.

err=3 Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where *x* is the solution for the current right-hand side and *s* scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of *z* are approximately 1.

The information for right-hand side *i*, where $1 \leq i \leq \text{nrhs}$, and type of error *err* is stored in:

- Column major layout: *err_bnds_comp*[(*err* - 1) * *nrhs* + *i* - 1].
- Row major layout: *err_bnds_comp*[*err* - 1 + (*i* - 1) * *n_err_bnds*]

params

Output parameter only if the input contains erroneous values, namely, in *params*[0], *params*[1], *params*[2]. In such a case, the corresponding elements of *params* are filled with default values on output.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, parameter *i* had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params*[2] = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that for column major layout *err_bnds_norm*[*j* - 1] = 0.0 or *err_bnds_comp*[*j* - 1] = 0.0; or for row major layout *err_bnds_norm*[(*j* - 1) * *n_err_bnds*] = 0.0 or *err_bnds_comp*[(*j* - 1) * *n_err_bnds*] = 0.0). See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

[Matrix Storage Schemes](#)

?sprfs

Refines the solution of a system of linear equations with a packed symmetric coefficient matrix and estimates the solution error.

Syntax

```
lapack_int LAPACKESsprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* ap, const float* afp, const lapack_int* ipiv, const float* b, lapack_int
ldb, float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKESdsprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* ap, const double* afp, const lapack_int* ipiv, const double* b,
lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );
```

```
lapack_int LAPACKEScsprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* ap, const lapack_complex_float* afp, const lapack_int*
ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKESzsprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* ap, const lapack_complex_double* afp, const lapack_int*
ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x,
lapack_int ldx, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a packed symmetric matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?spturf](#)
- call the solver routine [?spturs](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.

ap, afp, b, x

Arrays:

ap of size $\max(1, n(n+1)/2)$ contains the original packed matrix *A*, as supplied to `?spturf`.*afp* of size $\max(1, n(n+1)/2)$ contains the factored packed matrix *A*, as returned by `?spturf`.*b* of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the right-hand side matrix *B*.*x* of size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout contains the solution matrix *X*.*ldb*The leading dimension of *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.*ldx*The leading dimension of *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq \max(1, nrhs)$ for row major layout.*ipiv*Array, size at least $\max(1, n)$. The *ipiv* array, as returned by `?spturf`.

Output Parameters

*x*The refined solution matrix *X*.*ferr, berr*Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.If *info* = 0, the execution is successful.If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

`?hprfs`*Refines the solution of a system of linear equations with a packed complex Hermitian coefficient matrix and estimates the solution error.*

Syntax

```
lapack_int LAPACKE_chprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* ap, const lapack_complex_float* afp, const lapack_int*
ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_zhprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* ap, const lapack_complex_double* afp, const lapack_int*
ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x,
lapack_int ldx, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a packed complex Hermitian matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A) x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?hptrf](#)
- call the solver routine [?hptrs](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>ap, afp, b, x</i>	Arrays: <i>ap</i> max(1, $n(n + 1)/2$) contains the original packed matrix A , as supplied to ?hptrf . <i>afp</i> max(1, $n(n + 1)/2$) contains the factored packed matrix A , as returned by ?hptrf . <i>b</i> of size max(1, $ldb \cdot nrhs$) for column major layout and max(1, $ldb \cdot n$) for row major layout contains the right-hand side matrix B .

x of size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout contains the solution matrix X .

ldb The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ldx The leading dimension of x ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

ipiv Array, size at least $\max(1, n)$. The *ipiv* array, as returned by [?hptrf](#).

Output Parameters

x The refined solution matrix X .

ferr, berr Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is [?ssprfs](#)/[?dsprfs](#).

See Also

Matrix Storage Schemes

[?trrf](#)

Estimates the error in the solution of a system of linear equations with a triangular coefficient matrix.

Syntax

```
lapack_int LAPACKE_strrf( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const float* a, lapack_int lda, const float* b,
lapack_int ldb, const float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dtrrf( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const double* a, lapack_int lda, const double* b,
lapack_int ldb, const double* x, lapack_int ldx, double* ferr, double* berr );
```

```
lapack_int LAPACKE_ctrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* b, lapack_int ldb, const lapack_complex_float* x, lapack_int ldx,
float* ferr, float* berr );
```

```
lapack_int LAPACKE_ztrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* b, lapack_int ldb, const lapack_complex_double* x, lapack_int
ldx, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine estimates the errors in the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a triangular matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?trtrs](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>trans</i>	Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A*X = B$. If <i>trans</i> = 'T', the system has the form $A^T*X = B$. If <i>trans</i> = 'C', the system has the form $A^H*X = B$.
<i>diag</i>	Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array a .
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.

a, b, x

Arrays:

a(size $\max(1, lda*n)$) contains the upper or lower triangular matrix *A*, as specified by *uplo*.*b* of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the right-hand side matrix *B*.*x* of size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout contains the solution matrix *X*.*lda*The leading dimension of *a*; $lda \geq \max(1, n)$.*ldb*The leading dimension of *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.*ldx*The leading dimension of *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

*ferr, berr*Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.If *info* = 0, the execution is successful.If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

See Also

Matrix Storage Schemes

*?tprfs**Estimates the error in the solution of a system of linear equations with a packed triangular coefficient matrix.*

Syntax

```
lapack_int LAPACKE_stprfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const float* ap, const float* b, lapack_int ldb, const
float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dtpfrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const double* ap, const double* b, lapack_int ldb, const
double* x, lapack_int ldx, double* ferr, double* berr );
```

```
lapack_int LAPACKE_ctprfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const lapack_complex_float* ap, const
lapack_complex_float* b, lapack_int ldb, const lapack_complex_float* x, lapack_int ldx,
float* ferr, float* berr );
```

```
lapack_int LAPACKE_ztprfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const lapack_complex_double* ap, const
lapack_complex_double* b, lapack_int ldb, const lapack_complex_double* x, lapack_int
ldx, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine estimates the errors in the solution to a system of linear equations $A^*X = B$ or $A^T X = B$ or $A^H X = B$ with a packed triangular matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?tpttrs](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>trans</i>	Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A^*X = B$. If <i>trans</i> = 'T', the system has the form $A^T X = B$. If <i>trans</i> = 'C', the system has the form $A^H X = B$.
<i>diag</i>	Must be 'N' or 'U'. If <i>diag</i> = 'N', A is not a unit triangular matrix. If <i>diag</i> = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.

ap, b, x

Arrays:

ap of size $\max(1, n(n+1)/2)$ contains the upper or lower triangular matrix *A*, as specified by *uplo*.

b of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the right-hand side matrix *B*.

x of size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout contains the solution matrix *X*.

ldb

The leading dimension of *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ldx

The leading dimension of *x*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

ferr, berr

Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

See Also

Matrix Storage Schemes

?tbrfs

Estimates the error in the solution of a system of linear equations with a triangular band coefficient matrix.

Syntax

```
lapack_int LAPACKE_stbrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int kd, lapack_int nrhs, const float* ab, lapack_int ldab, const
float* b, lapack_int ldb, const float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_dtbrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int kd, lapack_int nrhs, const double* ab, lapack_int ldab, const
double* b, lapack_int ldb, const double* x, lapack_int ldx, double* ferr, double*
berr );
```

```

lapack_int LAPACKE_ctbrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int kd, lapack_int nrhs, const lapack_complex_float* ab,
lapack_int ldab, const lapack_complex_float* b, lapack_int ldb, const
lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_ztbrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int kd, lapack_int nrhs, const lapack_complex_double* ab,
lapack_int ldab, const lapack_complex_double* b, lapack_int ldb, const
lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );

```

Include Files

- mkl.h

Description

The routine estimates the errors in the solution to a system of linear equations $A^*X = B$ or $A^T * X = B$ or $A^H * X = B$ with a triangular band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?tbrfs](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>trans</i>	Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A^*X = B$. If <i>trans</i> = 'T', the system has the form $A^T * X = B$. If <i>trans</i> = 'C', the system has the form $A^H * X = B$.
<i>diag</i>	Must be 'N' or 'U'. If <i>diag</i> = 'N', A is not a unit triangular matrix. If <i>diag</i> = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <i>ab</i> .
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>kd</i>	The number of super-diagonals or sub-diagonals in the matrix A ; $kd \geq 0$.

<i>nrhs</i>	The number of right-hand sides; $nrhs \geq 0$.
<i>ab, b, x</i>	Arrays: $ab(\text{size max}(1, ldab*n))$ contains the upper or lower triangular matrix A , as specified by <i>uplo</i> , in band storage format. b of size $\text{max}(1, ldb*nrhs)$ for column major layout and $\text{max}(1, ldb*n)$ for row major layout contains the right-hand side matrix B . x of size $\text{max}(1, ldx*nrhs)$ for column major layout and $\text{max}(1, ldx*n)$ for row major layout contains the solution matrix X .
<i>ldab</i>	The leading dimension of the array <i>ab</i> ; $ldab \geq kd + 1$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \text{max}(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ldx</i>	The leading dimension of <i>x</i> ; $ldb \geq \text{max}(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>ferr, berr</i>	Arrays, size at least $\text{max}(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
-------------------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n*kd$ floating-point operations for real flavors or $8n*kd$ operations for complex flavors.

See Also

[Matrix Storage Schemes](#)

Matrix Inversion: LAPACK Computational Routines

It is seldom necessary to compute an explicit inverse of a matrix. In particular, do not attempt to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see [Routines for Solving Systems of Linear Equations](#)); this is more efficient and more accurate.

However, matrix inversion routines are provided for the rare occasions when an explicit inverse matrix is needed.

?getri

Computes the inverse of an LU-factored general matrix.

Syntax

```
lapack_int LAPACKE_sgetri (int matrix_layout , lapack_int n , float * a , lapack_int
lda , const lapack_int * ipiv );

lapack_int LAPACKE_dgetri (int matrix_layout , lapack_int n , double * a , lapack_int
lda , const lapack_int * ipiv );

lapack_int LAPACKE_cgetri (int matrix_layout , lapack_int n , lapack_complex_float *
a , lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_zgetri (int matrix_layout , lapack_int n , lapack_complex_double *
a , lapack_int lda , const lapack_int * ipiv );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a general matrix A . Before calling this routine, call [?getrf](#) to factorize A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	Array a (size $\max(1, lda * n)$) contains the factorization of the matrix A , as returned by ?getrf : $A = P * L * U$. The second dimension of a must be at least $\max(1, n)$.
<i>lda</i>	The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?getrf .

Output Parameters

<i>a</i>	Overwritten by the n -by- n matrix $\text{inv}(A)$.
----------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of the factor U is zero, U is singular, and the inversion could not be completed.

Application Notes

The computed inverse X satisfies the following error bound:

$$\|XA - I\| \leq c(n) \varepsilon \|X\| P \|L\| \|U\|,$$

where $c(n)$ is a modest linear function of n ; ε is the machine precision; I denotes the identity matrix; P , L , and U are the factors of the matrix factorization $A = P * L * U$.

The total number of floating-point operations is approximately $(4/3)n^3$ for real flavors and $(16/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

mkl_?getrnp

Computes the inverse of an LU-factored general matrix without pivoting.

Syntax

```
lapack_int LAPACKE_mkl_sgetrnp (int matrix_layout , lapack_int n , float * a ,
lapack_int lda );

lapack_int LAPACKE_mkl_dgetrnp (int matrix_layout , lapack_int n , double * a ,
lapack_int lda );

lapack_int LAPACKE_mkl_cgetrnp (int matrix_layout , lapack_int n ,
lapack_complex_float * a , lapack_int lda );

lapack_int LAPACKE_mkl_zgetrnp (int matrix_layout , lapack_int n ,
lapack_complex_double * a , lapack_int lda );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a general matrix A . Before calling this routine, call [mkl_?getrfnp](#) to factorize A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	Array a (size $\max(1, \text{lda} * n)$) contains the factorization of the matrix A , as returned by mkl_?getrfnp : $A = L * U$. The second dimension of a must be at least $\max(1, n)$.
<i>lda</i>	The leading dimension of a ; $\text{lda} \geq \max(1, n)$.

Output Parameters

<i>a</i>	Overwritten by the n -by- n matrix $\text{inv}(A)$.
----------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If $info = i$, the i -th diagonal element of the factor U is zero, U is singular, and the inversion could not be completed.

Application Notes

The total number of floating-point operations is approximately $(4/3)n^3$ for real flavors and $(16/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?potri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix using the Cholesky factorization.

Syntax

```
lapack_int LAPACKE_spotri (int matrix_layout , char uplo , lapack_int n , float * a ,
lapack_int lda );

lapack_int LAPACKE_dpotri (int matrix_layout , char uplo , lapack_int n , double * a ,
lapack_int lda );

lapack_int LAPACKE_cpotri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda );

lapack_int LAPACKE_zpotri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda );
```

Include Files

- mkl.h

Description

The routine computes the inverse $inv(A)$ of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A . Before calling this routine, call [?potrf](#) to factorize A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	Array $a(\text{size max}(1, lda * n))$. Contains the factorization of the matrix A , as returned by ?potrf .
<i>lda</i>	The leading dimension of a . $lda \geq \max(1, n)$.

Output Parameters

a Overwritten by the upper or lower triangle of the inverse of *A*.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(n) \varepsilon \kappa_2(A), \quad \|AX - I\|_2 \leq c(n) \varepsilon \kappa_2(A),$$

where $c(n)$ is a modest linear function of n , and ε is the machine precision; *I* denotes the identity matrix.

The 2-norm $\|A\|_2$ of a matrix *A* is defined by $\|A\|_2 = \max_{x \cdot x=1} (Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?pftri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix in RFP format using the Cholesky factorization.

Syntax

```
lapack_int LAPACKKE_spftri (int matrix_layout , char transr , char uplo , lapack_int n ,
float * a );
```

```
lapack_int LAPACKKE_dpftri (int matrix_layout , char transr , char uplo , lapack_int n ,
double * a );
```

```
lapack_int LAPACKKE_cpftri (int matrix_layout , char transr , char uplo , lapack_int n ,
lapack_complex_float * a );
```

```
lapack_int LAPACKKE_zpftri (int matrix_layout , char transr , char uplo , lapack_int n ,
lapack_complex_double * a );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric positive definite or, for complex data, Hermitian positive-definite matrix *A* using the Cholesky factorization:

$$A = U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} \quad \text{if } uplo='U'$$

$A = L * L^T$ for real data, $A = L * L^H$ for complex data if $uplo = 'L'$

Before calling this routine, call [?pftfrf](#) to factorize A .

The matrix A is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>transr</i>	Must be 'N', 'T' (for real data) or 'C' (for complex data). If <i>transr</i> = 'N', the Normal <i>transr</i> of RFP U (if <i>uplo</i> = 'U') or L (if <i>uplo</i> = 'L') is stored. If <i>transr</i> = 'T', the Transpose <i>transr</i> of RFP U (if <i>uplo</i> = 'U') or L (if <i>uplo</i> = 'L') is stored. If <i>transr</i> = 'C', the Conjugate-Transpose <i>transr</i> of RFP U (if <i>uplo</i> = 'U') or L (if <i>uplo</i> = 'L') is stored.
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', $A = U^T * U$ for real data or $A = U^H * U$ for complex data, and U is stored. If <i>uplo</i> = 'L', $A = L * L^T$ for real data or $A = L * L^H$ for complex data, and L is stored.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	Array, size $(n * (n + 1) / 2)$. The array <i>a</i> contains the factor U or L matrix A in the RFP format.

Output Parameters

<i>a</i>	The symmetric/Hermitian inverse of the original matrix in the same storage format.
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the (*i*, *i*) element of the factor U or L is zero, and the inverse could not be computed.

See Also

[Matrix Storage Schemes](#)

[?pptri](#)

Computes the inverse of a packed symmetric (Hermitian) positive-definite matrix using Cholesky factorization.

Syntax

```
lapack_int LAPACKE_spptri (int matrix_layout , char uplo , lapack_int n , float * ap );
lapack_int LAPACKE_dpptri (int matrix_layout , char uplo , lapack_int n , double *
ap );
lapack_int LAPACKE_cpptri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * ap );
lapack_int LAPACKE_zpptri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * ap );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A in *packed* form. Before calling this routine, call [?pptrf](#) to factorize A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular factor is stored in <i>ap</i> : If <i>uplo</i> = 'U', then the upper triangular factor is stored. If <i>uplo</i> = 'L', then the lower triangular factor is stored.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>ap</i>	Array, size at least $\max(1, n(n+1)/2)$. Contains the factorization of the packed matrix A , as returned by ?pptrf . The dimension <i>ap</i> must be at least $\max(1, n(n+1)/2)$.

Output Parameters

<i>ap</i>	Overwritten by the packed n -by- n matrix $\text{inv}(A)$.
-----------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

Application Notes

The computed inverse X satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(n) \varepsilon \kappa_2(A), \quad \|AX - I\|_2 \leq c(n) \varepsilon \kappa_2(A),$$

where $c(n)$ is a modest linear function of n , and ε is the machine precision; I denotes the identity matrix.

The 2-norm $\|A\|_2$ of a matrix A is defined by $\|A\|_2 = \max_{x \cdot x=1} (Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

[?sytri](#)

*Computes the inverse of a symmetric matrix using $U^*D^*U^T$ or $L^*D^*L^T$ Bunch-Kaufman factorization.*

Syntax

```
lapack_int LAPACKE_ssytri (int matrix_layout , char uplo , lapack_int n , float * a ,
lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_dsytri (int matrix_layout , char uplo , lapack_int n , double * a ,
lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_csytri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_zsytri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv );
```

Include Files

- `mk1.h`

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric matrix A . Before calling this routine, call [?sytrf](#) to factorize A .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the array a stores the Bunch-Kaufman factorization $A = U^*D^*U^T$. If <code>uplo = 'L'</code> , the array a stores the Bunch-Kaufman factorization $A = L^*D^*L^T$.
<code>n</code>	The order of the matrix A ; $n \geq 0$.

<i>a</i>	<i>a</i> (size max(1, <i>lda</i> * <i>n</i>)) contains the factorization of the matrix <i>A</i> , as returned by ?sytrf .
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ipiv</i>	Array, size at least max(1, <i>n</i>). The <i>ipiv</i> array, as returned by ?sytrf .

Output Parameters

<i>a</i>	Overwritten by the <i>n</i> -by- <i>n</i> matrix <i>inv</i> (<i>A</i>).
----------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D*U^T*P^T*X*P*U - I| \leq c(n)\varepsilon(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|D*L^T*P^T*X*P*L - I| \leq c(n)\varepsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here *c*(*n*) is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?hetri

*Computes the inverse of a complex Hermitian matrix using $U*D*U^H$ or $L*D*L^H$ Bunch-Kaufman factorization.*

Syntax

```
lapack_int LAPACKE_chetri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_zhetri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a complex Hermitian matrix A . Before calling this routine, call [?hetrf](#) to factorize A .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo</code> = 'U', the array a stores the Bunch-Kaufman factorization $A = U * D * U^H$. If <code>uplo</code> = 'L', the array a stores the Bunch-Kaufman factorization $A = L * D * L^H$.
<code>n</code>	The order of the matrix A ; $n \geq 0$.
<code>a,</code>	Array a (size $\max(1, lda * n)$) contains the factorization of the matrix A , as returned by ?hetrf . The second dimension of a must be at least $\max(1, n)$.
<code>lda</code>	The leading dimension of a ; $lda \geq \max(1, n)$.
<code>ipiv</code>	Array, size at least $\max(1, n)$. The <code>ipiv</code> array, as returned by ?hetrf .

Output Parameters

<code>a</code>	Overwritten by the n -by- n matrix $\text{inv}(A)$.
----------------	--

Return Values

This function returns a value `info`.

If `info` = 0, the execution is successful.

If `info` = $-i$, parameter i had an illegal value.

If `info` = i , the i -th diagonal element of D is zero, D is singular, and the inversion could not be completed.

Application Notes

The computed inverse X satisfies the following error bounds:

$$\|D * U^H * P^T * X * P * U - I\| \leq c(n) \varepsilon (\|D\| \|U^H\| P^T \|X\| P \|U\| + \|D\| \|D^{-1}\|)$$

for `uplo` = 'U', and

$$\|D * L^H * P^T * X * P * L - I\| \leq c(n) \varepsilon (\|D\| \|L^H\| P^T \|X\| P \|L\| + \|D\| \|D^{-1}\|)$$

for `uplo` = 'L'. Here $c(n)$ is a modest linear function of n , and ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(8/3) n^3$ for complex flavors.

The real counterpart of this routine is [?sytri](#).

See Also

Matrix Storage Schemes

?sytri2

Computes the inverse of a symmetric indefinite matrix through allocating memory and calling ?sytri2x.

Syntax

```
lapack_int LAPACKE_ssytri2 (int matrix_layout , char uplo , lapack_int n , float * a ,
lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_dsytri2 (int matrix_layout , char uplo , lapack_int n , double * a ,
lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_csytri2 (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_zsytri2 (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric indefinite matrix A using the factorization $A = U*D*U^T$ or $A = L*D*L^T$ computed by ?sytrf.

The ?sytri2 routine allocates a temporary buffer before calling ?sytri2x that actually computes the inverse.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the factorization $A = U*D*U^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the factorization $A = L*D*L^T$.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	Array <i>a</i> (size $\max(1, lda*n)$) contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as returned by ?sytrf.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. Details of the interchanges and the block structure of D as returned by ?sytrf.

Output Parameters

<i>a</i>	If <i>info</i> = 0, the symmetric inverse of the original matrix. If <i>uplo</i> = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced.
----------	---

If `uplo = 'L'`, the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i`, $D(i,i) = 0$; D is singular and its inversion could not be computed.

See Also

[?sytrf](#)

[?sytri2x](#)

Matrix Storage Schemes

[?hetri2](#)

Computes the inverse of a Hermitian indefinite matrix through allocating memory and calling [?hetri2x](#).

Syntax

```
lapack_int LAPACKE_chetri2 (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_zhetri2 (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv );
```

Include Files

- `mkl.h`

Description

The routine computes the inverse $\text{inv}(A)$ of a Hermitian indefinite matrix A using the factorization $A = U*D*U^H$ or $A = L*D*L^H$ computed by [?hetrf](#).

The [?hetri2](#) routine allocates a temporary buffer before calling [?hetri2x](#) that actually computes the inverse.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo = 'U'</code> , the array a stores the factorization $A = U*D*U^H$. If <code>uplo = 'L'</code> , the array a stores the factorization $A = L*D*L^H$.
<code>n</code>	The order of the matrix A ; $n \geq 0$.
<code>a</code>	Array a (size $\max(1, lda*n)$) contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as returned by ?sytrf .

<code>lda</code>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<code>ipiv</code>	Array, size at least $\max(1, n)$. Details of the interchanges and the block structure of <i>D</i> as returned by <code>?hetrf</code> .

Output Parameters

<code>a</code>	If <code>info = 0</code> , the inverse of the original matrix. If <code>uplo = 'U'</code> , the upper triangular part of the inverse is formed and the part of <i>A</i> below the diagonal is not referenced. If <code>uplo = 'L'</code> , the lower triangular part of the inverse is formed and the part of <i>A</i> above the diagonal is not referenced.
----------------	--

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, parameter *i* had an illegal value.

If `info = i`, $D(i,i) = 0$; *D* is singular and its inversion could not be computed.

See Also

[?hetrf](#)

[?hetri2x](#)

Matrix Storage Schemes

[?sytri2x](#)

Computes the inverse of a symmetric indefinite matrix after ?sytri2 allocates memory.

Syntax

```
lapack_int LAPACK_essytri2x (int matrix_layout , char uplo , lapack_int n , float * a ,
lapack_int lda , const lapack_int * ipiv , lapack_int nb );
```

```
lapack_int LAPACK_dsytri2x (int matrix_layout , char uplo , lapack_int n , double *
a , lapack_int lda , const lapack_int * ipiv , lapack_int nb );
```

```
lapack_int LAPACK_csytri2x (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv , lapack_int nb );
```

```
lapack_int LAPACK_zsytri2x (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv , lapack_int nb );
```

Include Files

- `mkl.h`

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric indefinite matrix *A* using the factorization $A = U^*D^*U^T$ or $A = L^*D^*L^T$ computed by `?sytrf`.

The `?sytri2x` actually computes the inverse after the `?sytri2` routine allocates memory before calling `?sytri2x`.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo</code> = 'U', the array a stores the factorization $A = U * D * U^T$. If <code>uplo</code> = 'L', the array a stores the factorization $A = L * D * L^T$.
<code>n</code>	The order of the matrix A ; $n \geq 0$.
<code>a</code>	Array a (size $\max(1, lda * n)$) contains the nb (block size) diagonal matrix D and the multipliers used to obtain the factor U or L as returned by <code>?sytrf</code> . The second dimension of a must be at least $\max(1, n)$.
<code>lda</code>	The leading dimension of a ; $lda \geq \max(1, n)$.
<code>ipiv</code>	Array, size at least $\max(1, n)$. Details of the interchanges and the nb structure of D as returned by <code>?sytrf</code> .
<code>nb</code>	Block size.

Output Parameters

<code>a</code>	If <code>info</code> = 0, the symmetric inverse of the original matrix. If <code>info</code> = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If <code>info</code> = 'L', the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
----------------	---

Return Values

This function returns a value `info`.

If `info` = 0, the execution is successful.

If `info` = $-i$, parameter i had an illegal value.

If `info` = i , D_{ii} = 0; D is singular and its inversion could not be computed.

See Also

[?sytrf](#)

[?sytri2](#)

Matrix Storage Schemes

[?hetri2x](#)

Computes the inverse of a Hermitian indefinite matrix after `?hetri2` allocates memory.

Syntax

```
lapack_int LAPACKE_chetri2x (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv , lapack_int nb );

lapack_int LAPACKE_zhetri2x (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv , lapack_int nb );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a Hermitian indefinite matrix A using the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ computed by ?hetrf.

The ?hetri2x actually computes the inverse after the ?hetri2 routine allocates memory before calling ?hetri2x.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the factorization $A = U^*D^*U^H$. If <i>uplo</i> = 'L', the array <i>a</i> stores the factorization $A = L^*D^*L^H$.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	Arrays <i>a</i> (size $\max(1, lda*n)$) contains the <i>nb</i> (block size) diagonal matrix D and the multipliers used to obtain the factor U or L as returned by ?hetrf.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. Details of the interchanges and the <i>nb</i> structure of D as returned by ?hetrf.
<i>nb</i>	Block size.

Output Parameters

<i>a</i>	If <i>info</i> = 0, the symmetric inverse of the original matrix. If <i>info</i> = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If <i>info</i> = 'L', the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
----------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, $D_{ii} = 0$; *D* is singular and its inversion could not be computed.

See Also

[?hetrf](#)

[?hetri2](#)

Matrix Storage Schemes

[?sytri_3](#)

Computes the inverse of a real or complex symmetric matrix.

```
lapack_int LAPACKE_ssytri_3 (int matrix_layout, char uplo, lapack_int n, float * A,
lapack_int lda, const float * e, const lapack_int * ipiv);

lapack_int LAPACKE_dsytri_3 (int matrix_layout, char uplo, lapack_int n, double * A,
lapack_int lda, const double * e, const lapack_int * ipiv);

lapack_int LAPACKE_csytri_3 (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e, const
lapack_int * ipiv);

lapack_int LAPACKE_zsytri_3 (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e, const
lapack_int * ipiv);
```

Description

[?sytri_3](#) computes the inverse of a real or complex symmetric matrix *A* using the factorization computed by [?sytrf_rk](#): $A = P * U * D * (U^T)^*(P^T)$ or $A = P * L * D * (L^T)^*(P^T)$, where *U* (or *L*) is a unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of *U* (or *L*), *P* is a permutation matrix, P^T is the transpose of *P*, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

[?sytri_3](#) sets the leading dimension of the workspace before calling [?sytri_3x](#), which actually computes the inverse. This is the blocked version of the algorithm, calling Level-3 BLAS.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. <ul style="list-style-type: none"> = 'U': The upper triangle of <i>A</i> is stored. = 'L': The lower triangle of <i>A</i> is stored.
<i>n</i>	The order of the matrix <i>A</i> . $n \geq 0$.
<i>A</i>	Array of size $\max(1, lda * n)$. On entry, diagonal of the block diagonal matrix <i>D</i> and factors <i>U</i> or <i>L</i> as computed by ?sytrf_rk : <ul style="list-style-type: none"> Only diagonal elements of the symmetric block diagonal matrix <i>D</i> on the diagonal of <i>A</i>; that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of <i>D</i> should be provided on entry in array <i>e</i>. <p>—and—</p>

- If `uplo = 'U'`, factor U in the superdiagonal part of A. If `uplo = 'L'`, factor L in the subdiagonal part of A.

`lda`

The leading dimension of the array `A`.

`e`

Array of size n . On entry, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix `D` with 1-by-1 or 2-by-2 diagonal blocks. If `uplo = 'U'`, $e(i) = D(i-1,i)$, $i=2:N$, and $e(1)$ is not referenced. If `uplo = 'L'`, $e(i) = D(i+1,i)$, $i=1:N-1$, and $e(n)$ is not referenced.

NOTE For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e[k-1]$ is not referenced in both the `uplo = 'U'` and `uplo = 'L'` cases.

`ipiv`

Array of size n . Details of the interchanges and the block structure of `D` as determined by `?sytrf_rk`.

Output Parameters

`A`

On exit, if `info = 0`, the symmetric inverse of the original matrix. If `uplo = 'U'`, the upper triangular part of the inverse is formed and the part of `A` below the diagonal is not referenced. If `uplo = 'L'`, the lower triangular part of the inverse is formed and the part of `A` above the diagonal is not referenced.

Return Values

This function returns a value `info`.

= 0: Successful exit.

< 0: If `info = -i`, the i^{th} argument had an illegal value.

> 0: If `info = i`, $D(i,i) = 0$; the matrix is singular and its inverse could not be computed.

?hetri_3

Computes the inverse of a complex Hermitian matrix using the factorization computed by ?hetrf_rk.

```
lapack_int LAPACKC_chetri_3 (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e, const
lapack_int * ipiv);
```

```
lapack_int LAPACKC_zhetri_3 (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e, const
lapack_int * ipiv);
```

Description

`?hetri_3` computes the inverse of a complex Hermitian matrix `A` using the factorization computed by `?hetrf_rk`: $A = P*U*D*(U^H)*(P^T)$ or $A = P*L*D*(L^H)*(P^T)$, where `U` (or `L`) is a unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of `U` (or `L`), `P` is a permutation matrix, P^T is the transpose of `P`, and `D` is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

`?hetri_3` sets the leading dimension of the workspace before calling `?hetri_3x`, which actually computes the inverse.

This is the blocked version of the algorithm, calling Level-3 BLAS.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies whether the details of the factorization are stored as an upper or lower triangular matrix. <ul style="list-style-type: none"> = 'U': The upper triangle of A is stored. = 'L': The lower triangle of A is stored.
<i>n</i>	The order of the matrix A. $n \geq 0$.
<i>A</i>	Array of size $\max(1, lda * n)$. On entry, diagonal of the block diagonal matrix D and factor U or L as computed by ?hetrf_rk: <ul style="list-style-type: none"> Only diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A; that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of D should be provided on entry in array <i>e</i>. If <i>uplo</i> = 'U', factor U in the superdiagonal part of A. If <i>uplo</i> = 'L', factor L is the subdiagonal part of A.
<i>lda</i>	The leading dimension of the array A.
<i>e</i>	Array of size <i>n</i> . On entry, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If <i>uplo</i> = 'U', $e(i) = D(i-1,i)$, $i=2:N$, and $e(1)$ is not referenced. If <i>uplo</i> = 'L', $e(i) = D(i+1,i)$, $i=1:N-1$, and $e(n)$ is not referenced.

NOTE For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e[k-1]$ is not referenced in both the *uplo* = 'U' and *uplo* = 'L' cases.

<i>ipiv</i>	Array of size <i>n</i> . Details of the interchanges and the block structure of D as determined by ?hetrf_rk.
-------------	---

Output Parameters

<i>A</i>	On exit, if <i>info</i> = 0, the Hermitian inverse of the original matrix. If <i>uplo</i> = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If <i>uplo</i> = 'L', the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
----------	--

Return Values

This function returns a value *info*.

= 0: Successful exit.

< 0: If *info* = -*i*, the *i*th argument had an illegal value.

> 0: If *info* = *i*, $D(i,i) = 0$; the matrix is singular and its inverse could not be computed.

?sptri

Computes the inverse of a symmetric matrix using $U*D*U^T$ or $L*D*L^T$ Bunch-Kaufman factorization of matrix in packed storage.

Syntax

```
lapack_int LAPACK_essptri (int matrix_layout , char uplo , lapack_int n , float * ap ,
const lapack_int * ipiv );

lapack_int LAPACK_dsptri (int matrix_layout , char uplo , lapack_int n , double * ap ,
const lapack_int * ipiv );

lapack_int LAPACK_csptri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * ap , const lapack_int * ipiv );

lapack_int LAPACK_zsptri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * ap , const lapack_int * ipiv );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a packed symmetric matrix A . Before calling this routine, call [?spturf](#) to factorize A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the Bunch-Kaufman factorization $A = U*D*U^T$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the Bunch-Kaufman factorization $A = L*D*L^T$.
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>ap</i>	Arrays <i>ap</i> (size $\max(1, n(n+1)/2)$) contains the factorization of the matrix A , as returned by ?spturf .
<i>ipiv</i>	Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?spturf .

Output Parameters

<i>ap</i>	Overwritten by the matrix $\text{inv}(A)$ in packed form.
-----------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If $info = -i$, parameter i had an illegal value.

If $info = i$, the i -th diagonal element of D is zero, D is singular, and the inversion could not be completed.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|D*U^T*P^T*X*P*U - I| \leq c(n)\epsilon(|D||U^T|P^T|X|P|U| + |D||D^{-1}|)$$

for $uplo = 'U'$, and

$$|D*L^T*P^T*X*P*L - I| \leq c(n)\epsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for $uplo = 'L'$. Here $c(n)$ is a modest linear function of n , and ϵ is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

?hptri

*Computes the inverse of a complex Hermitian matrix using $U*D*U^H$ or $L*D*L^H$ Bunch-Kaufman factorization of matrix in packed storage.*

Syntax

```
lapack_int LAPACKE_chptri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * ap , const lapack_int * ipiv );

lapack_int LAPACKE_zhptri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * ap , const lapack_int * ipiv );
```

Include Files

- mkl.h

Description

The routine computes the inverse $inv(A)$ of a complex Hermitian matrix A using packed storage. Before calling this routine, call [?hptrf](#) to factorize A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array ap stores the packed Bunch-Kaufman factorization $A = U*D*U^H$. If $uplo = 'L'$, the array ap stores the packed Bunch-Kaufman factorization $A = L*D*L^H$.
<i>n</i>	The order of the matrix A ; $n \geq 0$.

ap Array *ap* (size $\max(1, n(n+1)/2)$) contains the factorization of the matrix *A*, as returned by [?hptrf](#).

ipiv Array, size at least $\max(1, n)$.
The *ipiv* array, as returned by [?hptrf](#).

Output Parameters

ap Overwritten by the matrix $\text{inv}(A)$.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D*U^H*P^T*X*P*U - I| \leq c(n)\varepsilon(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for *uplo* = 'U', and

$$|D*L^H*P^T*X*P*L - I| \leq c(n)\varepsilon(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for *uplo* = 'L'. Here $c(n)$ is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(8/3)n^3$.

The real counterpart of this routine is [?sptri](#).

See Also

Matrix Storage Schemes

[?trtri](#)

Computes the inverse of a triangular matrix.

Syntax

```
lapack_int LAPACKKE_strtri (int matrix_layout , char uplo , char diag , lapack_int n ,
float * a , lapack_int lda );
```

```
lapack_int LAPACKKE_dtrtri (int matrix_layout , char uplo , char diag , lapack_int n ,
double * a , lapack_int lda );
```

```
lapack_int LAPACKKE_ctrtri (int matrix_layout , char uplo , char diag , lapack_int n ,
lapack_complex_float * a , lapack_int lda );
```

```
lapack_int LAPACKKE_ztrtri (int matrix_layout , char uplo , char diag , lapack_int n ,
lapack_complex_double * a , lapack_int lda );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a triangular matrix A .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <code>uplo</code> = 'U', then A is upper triangular. If <code>uplo</code> = 'L', then A is lower triangular.
<code>diag</code>	Must be 'N' or 'U'. If <code>diag</code> = 'N', then A is not a unit triangular matrix. If <code>diag</code> = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array a .
<code>n</code>	The order of the matrix A ; $n \geq 0$.
<code>a</code>	Array: . Contains the matrix A .
<code>lda</code>	The first dimension of a ; $lda \geq \max(1, n)$.

Output Parameters

<code>a</code>	Overwritten by the matrix $\text{inv}(A)$.
----------------	---

Return Values

This function returns a value `info`.

If `info` = 0, the execution is successful.

If `info` = $-i$, parameter i had an illegal value.

If `info` = i , the i -th diagonal element of A is zero, A is singular, and the inversion could not be completed.

Application Notes

The computed inverse X satisfies the following error bounds:

$$\|XA - I\| \leq c(n)\varepsilon \|X\| \|A\|$$

$$\|XA - I\| \leq c(n)\varepsilon \|A^{-1}\| \|A\| \|X\|,$$

where $c(n)$ is a modest linear function of n ; ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?tftri

Computes the inverse of a triangular matrix stored in the Rectangular Full Packed (RFP) format.

Syntax

```
lapack_int LAPACKE_stftri (int matrix_layout , char transr , char uplo , char diag ,
lapack_int n , float * a );

lapack_int LAPACKE_dtftri (int matrix_layout , char transr , char uplo , char diag ,
lapack_int n , double * a );

lapack_int LAPACKE_ctftri (int matrix_layout , char transr , char uplo , char diag ,
lapack_int n , lapack_complex_float * a );

lapack_int LAPACKE_ztftri (int matrix_layout , char transr , char uplo , char diag ,
lapack_int n , lapack_complex_double * a );
```

Include Files

- mkl.h

Description

Computes the inverse of a triangular matrix *A* stored in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

This is the block version of the algorithm, calling Level 3 BLAS.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>transr</i>	Must be 'N', 'T' (for real data) or 'C' (for complex data). If <i>transr</i> = 'N', the Normal <i>transr</i> of RFP <i>A</i> is stored. If <i>transr</i> = 'T', the Transpose <i>transr</i> of RFP <i>A</i> is stored. If <i>transr</i> = 'C', the Conjugate-Transpose <i>transr</i> of RFP <i>A</i> is stored.
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of RFP <i>A</i> is stored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix <i>A</i> . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix <i>A</i> .
<i>diag</i>	Must be 'N' or 'U'. If <i>diag</i> = 'N', then <i>A</i> is not a unit triangular matrix. If <i>diag</i> = 'U', <i>A</i> is unit triangular: diagonal elements of <i>A</i> are assumed to be 1 and not referenced in the array <i>a</i> .
<i>n</i>	The order of the matrix <i>A</i> ; $n \geq 0$.

a Array, size $\max(1, n*(n + 1)/2)$. The array *a* contains the matrix *A* in the RFP format.

Output Parameters

a The (triangular) inverse of the original matrix in the same storage format.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, $A_{i,i}$ is exactly zero. The triangular matrix is singular and its inverse cannot be computed.

See Also

Matrix Storage Schemes

?tptri

Computes the inverse of a triangular matrix using packed storage.

Syntax

```
lapack_int LAPACKE_stptri (int matrix_layout , char uplo , char diag , lapack_int n ,
float * ap );
```

```
lapack_int LAPACKE_dtptri (int matrix_layout , char uplo , char diag , lapack_int n ,
double * ap );
```

```
lapack_int LAPACKE_ctptri (int matrix_layout , char uplo , char diag , lapack_int n ,
lapack_complex_float * ap );
```

```
lapack_int LAPACKE_ztptri (int matrix_layout , char uplo , char diag , lapack_int n ,
lapack_complex_double * ap );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a packed triangular matrix *A*.

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo Must be 'U' or 'L'.

Indicates whether *A* is upper or lower triangular:

If *uplo* = 'U', then *A* is upper triangular.

If *uplo* = 'L', then *A* is lower triangular.

<i>diag</i>	Must be 'N' or 'U'. If <i>diag</i> = 'N', then <i>A</i> is not a unit triangular matrix. If <i>diag</i> = 'U', <i>A</i> is unit triangular: diagonal elements of <i>A</i> are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	The order of the matrix <i>A</i> ; $n \geq 0$.
<i>ap</i>	Array, size at least $\max(1, n(n+1)/2)$. Contains the packed triangular matrix <i>A</i> .

Output Parameters

<i>ap</i>	Overwritten by the packed <i>n</i> -by- <i>n</i> matrix <code>inv(A)</code> .
-----------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is zero, *A* is singular, and the inversion could not be completed.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$\|XA - I\| \leq c(n)\varepsilon \|X\| \|A\|$$

$$\|X - A^{-1}\| \leq c(n)\varepsilon \|A^{-1}\| \|A\| \|X\|,$$

where $c(n)$ is a modest linear function of *n*; ε is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

See Also

Matrix Storage Schemes

Matrix Equilibration: LAPACK Computational Routines

Routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

?geequ

Computes row and column scaling factors intended to equilibrate a general matrix and reduce its condition number.

Syntax

```
lapack_int LAPACKGe_sgeequ( int matrix_layout, lapack_int m, lapack_int n, const float*
a, lapack_int lda, float* r, float* c, float* rowcnd, float* colcnd, float* amax );
```

```
lapack_int LAPACKGe_dgeequ( int matrix_layout, lapack_int m, lapack_int n, const double*
a, lapack_int lda, double* r, double* c, double* rowcnd, double* colcnd, double* amax );
```

```
lapack_int LAPACKE_cgeequ( int matrix_layout, lapack_int m, lapack_int n, const
lapack_complex_float* a, lapack_int lda, float* r, float* c, float* rowcnd, float*
colcnd, float* amax );

lapack_int LAPACKE_zgeequ( int matrix_layout, lapack_int m, lapack_int n, const
lapack_complex_double* a, lapack_int lda, double* r, double* c, double* rowcnd, double*
colcnd, double* amax );
```

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate an m -by- n matrix A and reduce its condition number. The output array r returns the row scale factors and the array c the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r[i-1]*a_{ij}*c[j-1]$ have absolute value 1.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows of the matrix A ; $m \geq 0$.
<i>n</i>	The number of columns of the matrix A ; $n \geq 0$.
<i>a</i>	Array: size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout. Contains the m -by- n matrix A whose equilibration factors are to be computed.
<i>lda</i>	The leading dimension of a ; $lda \geq \max(1, m)$.

Output Parameters

<i>r, c</i>	Arrays: r (size m), c (size n). If $info = 0$, or $info > m$, the array r contains the row scale factors of the matrix A . If $info = 0$, the array c contains the column scale factors of the matrix A .
<i>rowcnd</i>	If $info = 0$ or $info > m$, <i>rowcnd</i> contains the ratio of the smallest $r[i]$ to the largest $r[i]$.
<i>colcnd</i>	If $info = 0$, <i>colcnd</i> contains the ratio of the smallest $c[i]$ to the largest $c[i]$.
<i>amax</i>	Absolute value of the largest element of the matrix A .

Return Values

This function returns a value *info*.

If $info = 0$, the execution is successful.

If `info = -i`, parameter `i` had an illegal value.

If `info = i`, $i > 0$, and

$i \leq m$, the i -th row of A is exactly zero;

$i > m$, the $(i-m)$ th column of A is exactly zero.

Application Notes

All the components of r and c are restricted to be between `SMLNUM` = smallest safe number and `BIGNUM` = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

`SMLNUM` and `BIGNUM` are parameters representing machine precision. You can use the `?lamch` routines to compute them. For example, compute single precision values of `SMLNUM` and `BIGNUM` as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

If `rowcnd` ≥ 0.1 and `amax` is neither too large nor too small, it is not worth scaling by r .

If `colcnd` ≥ 0.1 , it is not worth scaling by c .

If `amax` is very close to `SMLNUM` or very close to `BIGNUM`, the matrix A should be scaled.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?geequb

Computes row and column scaling factors restricted to a power of radix to equilibrate a general matrix and reduce its condition number.

Syntax

```
lapack_int LAPACKE_sgeequb( int matrix_layout, lapack_int m, lapack_int n, const float*
a, lapack_int lda, float* r, float* c, float* rowcnd, float* colcnd, float* amax );
```

```
lapack_int LAPACKE_dgeequb( int matrix_layout, lapack_int m, lapack_int n, const
double* a, lapack_int lda, double* r, double* c, double* rowcnd, double* colcnd, double*
amax );
```

```
lapack_int LAPACKE_cgeequb( int matrix_layout, lapack_int m, lapack_int n, const
lapack_complex_float* a, lapack_int lda, float* r, float* c, float* rowcnd, float*
colcnd, float* amax );
```

```
lapack_int LAPACKE_zgeequb( int matrix_layout, lapack_int m, lapack_int n, const
lapack_complex_double* a, lapack_int lda, double* r, double* c, double* rowcnd, double*
colcnd, double* amax );
```

Include Files

- `mkl.h`

Description

The routine computes row and column scalings intended to equilibrate an m -by- n general matrix A and reduce its condition number. The output array r returns the row scale factors and the array c - the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{i,j} = r[i-1] * a_{i,j} * c[j-1]$ have an absolute value of at most the radix.

$r[i-1]$ and $c[j-1]$ are restricted to be a power of the radix between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of a but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the `?lamch` routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

This routine differs from `?geequ` by restricting the scaling factors to a power of the radix. Except for over- and underflow, scaling by these factors introduces no additional rounding errors. However, the scaled entries' magnitudes are no longer equal to approximately 1 but lie between $\sqrt{\text{radix}}$ and $1/\sqrt{\text{radix}}$.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>m</code>	The number of rows of the matrix A ; $m \geq 0$.
<code>n</code>	The number of columns of the matrix A ; $n \geq 0$.
<code>a</code>	Array: size $\max(1, lda * n)$ for column major layout and $\max(1, lda * m)$ for row major layout. Contains the m -by- n matrix A whose equilibration factors are to be computed.
<code>lda</code>	The leading dimension of a ; $lda \geq \max(1, m)$.

Output Parameters

<code>r, c</code>	Arrays: $r(m)$, $c(n)$. If $info = 0$, or $info > m$, the array r contains the row scale factors for the matrix A . If $info = 0$, the array c contains the column scale factors for the matrix A .
<code>rowcnd</code>	If $info = 0$ or $info > m$, <code>rowcnd</code> contains the ratio of the smallest $r[i]$ to the largest $r[i]$. If <code>rowcnd</code> ≥ 0.1 , and <code>amax</code> is neither too large nor too small, it is not worth scaling by r .
<code>colcnd</code>	If $info = 0$, <code>colcnd</code> contains the ratio of the smallest $c[i]$ to the largest $c[i]$. If <code>colcnd</code> ≥ 0.1 , it is not worth scaling by c .
<code>amax</code>	Absolute value of the largest element of the matrix A . If <code>amax</code> is very close to $SMLNUM$ or very close to $BIGNUM$, the matrix should be scaled.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, parameter `i` had an illegal value.

If `info = i`, $i > 0$, and

$i \leq m$, the i -th row of A is exactly zero;

$i > m$, the $(i-m)$ -th column of A is exactly zero.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?gbequ

Computes row and column scaling factors intended to equilibrate a banded matrix and reduce its condition number.

Syntax

```
lapack_int LAPACKE_sgbequ( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const float* ab, lapack_int ldab, float* r, float* c, float* rowcnd,
float* colcnd, float* amax );
```

```
lapack_int LAPACKE_dgbequ( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const double* ab, lapack_int ldab, double* r, double* c, double*
rowcnd, double* colcnd, double* amax );
```

```
lapack_int LAPACKE_cgbequ( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const lapack_complex_float* ab, lapack_int ldab, float* r, float* c,
float* rowcnd, float* colcnd, float* amax );
```

```
lapack_int LAPACKE_zgbequ( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const lapack_complex_double* ab, lapack_int ldab, double* r, double*
c, double* rowcnd, double* colcnd, double* amax );
```

Include Files

- `mk1.h`

Description

The routine computes row and column scalings intended to equilibrate an m -by- n band matrix A and reduce its condition number. The output array `r` returns the row scale factors and the array `c` the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r[i-1]*a_{ij}*c[j-1]$ have absolute value 1.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>m</code>	The number of rows of the matrix A ; $m \geq 0$.
<code>n</code>	The number of columns of the matrix A ; $n \geq 0$.
<code>kl</code>	The number of subdiagonals within the band of A ; $kl \geq 0$.

<i>ku</i>	The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>ab</i>	Array, size $\max(1, ldab*n)$ for column major layout and $\max(1, ldab*m)$ for row major layout. Contains the original band matrix <i>A</i> .
<i>ldab</i>	The leading dimension of <i>ab</i> ; $ldab \geq kl + ku + 1$.

Output Parameters

<i>r</i> , <i>c</i>	Arrays: <i>r</i> (size <i>m</i>), <i>c</i> (size <i>n</i>). If <i>info</i> = 0, or <i>info</i> > <i>m</i> , the array <i>r</i> contains the row scale factors of the matrix <i>A</i> . If <i>info</i> = 0, the array <i>c</i> contains the column scale factors of the matrix <i>A</i> .
<i>rowcnd</i>	If <i>info</i> = 0 or <i>info</i> > <i>m</i> , <i>rowcnd</i> contains the ratio of the smallest <i>r</i> [<i>i</i>] to the largest <i>r</i> [<i>i</i>].
<i>colcnd</i>	If <i>info</i> = 0, <i>colcnd</i> contains the ratio of the smallest <i>c</i> [<i>i</i>] to the largest <i>c</i> [<i>i</i>].
<i>amax</i>	Absolute value of the largest element of the matrix <i>A</i> .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i* and

$i \leq m$, the *i*-th row of *A* is exactly zero;

$i > m$, the (*i*-*m*)th column of *A* is exactly zero.

Application Notes

All the components of *r* and *c* are restricted to be between *SMLNUM* = smallest safe number and *BIGNUM* = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of *A* but works well in practice.

SMLNUM and *BIGNUM* are parameters representing machine precision. You can use the [?lamch](#) routines to compute them. For example, compute single precision values of *SMLNUM* and *BIGNUM* as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

If *rowcnd* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r*.

If *colcnd* ≥ 0.1 , it is not worth scaling by *c*.

If *amax* is very close to *SMLNUM* or very close to *BIGNUM*, the matrix *A* should be scaled.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?gbequb

Computes row and column scaling factors restricted to a power of radix to equilibrate a banded matrix and reduce its condition number.

Syntax

```
lapack_int LAPACKE_sgbequb( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const float* ab, lapack_int ldab, float* r, float* c, float* rowcnd,
float* colcnd, float* amax );
```

```
lapack_int LAPACKE_dgbequb( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const double* ab, lapack_int ldab, double* r, double* c, double*
rowcnd, double* colcnd, double* amax );
```

```
lapack_int LAPACKE_cgbequb( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const lapack_complex_float* ab, lapack_int ldab, float* r, float* c,
float* rowcnd, float* colcnd, float* amax );
```

```
lapack_int LAPACKE_zgbequb( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const lapack_complex_double* ab, lapack_int ldab, double* r, double*
c, double* rowcnd, double* colcnd, double* amax );
```

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate an m -by- n banded matrix A and reduce its condition number. The output array r returns the row scale factors and the array c - the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{i,j} = r[i-1] * a_{i,j} * c[j-1]$ have an absolute value of at most the radix.

$r[i]$ and $c[j]$ are restricted to be a power of the radix between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of a but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the ?lamch routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

This routine differs from ?gbequ by restricting the scaling factors to a power of the radix. Except for over- and underflow, scaling by these factors introduces no additional rounding errors. However, the scaled entries' magnitudes are no longer equal to approximately 1 but lie between $\sqrt{\text{radix}}$ and $1/\sqrt{\text{radix}}$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows of the matrix A ; $m \geq 0$.
<i>n</i>	The number of columns of the matrix A ; $n \geq 0$.
<i>kl</i>	The number of subdiagonals within the band of A ; $kl \geq 0$.

<i>ku</i>	The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>ab</i>	Array: size $\max(1, ldab*n)$ for column major layout and $\max(1, ldab*m)$ for row major layout
<i>ldab</i>	The leading dimension of <i>a</i> ; $ldab \geq \max(1, m)$.

Output Parameters

<i>r, c</i>	Arrays: <i>r</i> (size <i>m</i>), <i>c</i> (size <i>n</i>). If <i>info</i> = 0, or <i>info</i> > <i>m</i> , the array <i>r</i> contains the row scale factors for the matrix <i>A</i> . If <i>info</i> = 0, the array <i>c</i> contains the column scale factors for the matrix <i>A</i> .
<i>rowcnd</i>	If <i>info</i> = 0 or <i>info</i> > <i>m</i> , <i>rowcnd</i> contains the ratio of the smallest <i>r</i> (<i>i</i>) to the largest <i>r</i> (<i>i</i>). If <i>rowcnd</i> ≥ 0.1 , and <i>amax</i> is neither too large nor too small, it is not worth scaling by <i>r</i> .
<i>colcnd</i>	If <i>info</i> = 0, <i>colcnd</i> contains the ratio of the smallest <i>c</i> [<i>i</i>] to the largest <i>c</i> [<i>i</i>]. If <i>colcnd</i> ≥ 0.1 , it is not worth scaling by <i>c</i> .
<i>amax</i>	Absolute value of the largest element of the matrix <i>A</i> . If <i>amax</i> is very close to SMLNUM or BIGNUM, the matrix should be scaled.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

$i \leq m$, the *i*-th row of *A* is exactly zero;

$i > m$, the (*i*-*m*)-th column of *A* is exactly zero.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

```
lapack_int LAPACKE_spoequ( int matrix_layout, lapack_int n, const float* a, lapack_int lda, float* s, float* scnd, float* amax );
```

```
lapack_int LAPACKE_dpoequ( int matrix_layout, lapack_int n, const double* a, lapack_int lda, double* s, double* scnd, double* amax );
```

```
lapack_int LAPACKE_cpoequ( int matrix_layout, lapack_int n, const lapack_complex_float* a, lapack_int lda, float* s, float* scnd, float* amax );
```

```
lapack_int LAPACKE_zpoequ( int matrix_layout, lapack_int n, const
lapack_complex_double* a, lapack_int lda, double* s, double* scond, double* amax );
```

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix A and reduce its condition number (with respect to the two-norm). The output array s returns scale factors such that contains

$$1/\sqrt{a_{i,i}}$$

These factors are chosen so that the scaled matrix B with elements $B_{i,j}=s[i-1]*A_{i,j}*s[j-1]$ has diagonal elements equal to 1.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	Array: size $\max(1, lda*n)$. Contains the n -by- n symmetric or Hermitian positive definite matrix A whose scaling factors are to be computed. Only the diagonal elements of A are referenced.
<i>lda</i>	The leading dimension of a ; $lda \geq \max(1, n)$.

Output Parameters

<i>s</i>	Array, size n . If <i>info</i> = 0, the array s contains the scale factors for A .
<i>scond</i>	If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest $s[i]$ to the largest $s[i]$.
<i>amax</i>	Absolute value of the largest element of the matrix A .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = $-i$, parameter i had an illegal value.

If *info* = i , the i -th diagonal element of A is nonpositive.

Application Notes

If $s_{cond} \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by s .

If $amax$ is very close to `SMLNUM` or very close to `BIGNUM`, the matrix A should be scaled.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?poequb

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

```
lapack_int LAPACKE_spoequb( int matrix_layout, lapack_int n, const float* a, lapack_int
lda, float* s, float* scond, float* amax );
```

```
lapack_int LAPACKE_dpoequb( int matrix_layout, lapack_int n, const double* a,
lapack_int lda, double* s, double* scond, double* amax );
```

```
lapack_int LAPACKE_cpoequb( int matrix_layout, lapack_int n, const
lapack_complex_float* a, lapack_int lda, float* s, float* scond, float* amax );
```

```
lapack_int LAPACKE_zpoequb( int matrix_layout, lapack_int n, const
lapack_complex_double* a, lapack_int lda, double* s, double* scond, double* amax );
```

Include Files

- `mkl.h`

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix A and reduce its condition number (with respect to the two-norm).

These factors are chosen so that the scaled matrix B with elements $B_{i,j} = s[i-1] * A_{i,j} * s[j-1]$ has diagonal elements equal to 1. $s[i-1]$ is a power of two nearest to, but not exceeding $1/\sqrt{A_{i,i}}$.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	Array: size $\max(1, lda * n)$. Contains the n -by- n symmetric or Hermitian positive definite matrix A whose scaling factors are to be computed. Only the diagonal elements of A are referenced.
<i>lda</i>	The leading dimension of a ; $lda \geq \max(1, m)$.

Output Parameters

<i>s</i>	Array, size (<i>n</i>). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> [<i>i</i>] to the largest <i>s</i> [<i>i</i>]. If <i>scond</i> ≥ 0.1, and <i>amax</i> is neither too large nor too small, it is not worth scaling by <i>s</i> .
<i>amax</i>	Absolute value of the largest element of the matrix <i>A</i> . If <i>amax</i> is very close to SMLNUM or BIGNUM, the matrix should be scaled.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?ppequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix in packed storage and reduce its condition number.

Syntax

```
lapack_int LAPACKE_sppequ( int matrix_layout, char uplo, lapack_int n, const float* ap,
float* s, float* acond, float* amax );
```

```
lapack_int LAPACKE_dppequ( int matrix_layout, char uplo, lapack_int n, const double*
ap, double* s, double* acond, double* amax );
```

```
lapack_int LAPACKE_cppequ( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* ap, float* s, float* acond, float* amax );
```

```
lapack_int LAPACKE_zppequ( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* ap, double* s, double* acond, double* amax );
```

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix *A* in packed storage and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors such that contains

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements $b_{ij}=s[i-1]*a_{ij}*s[j-1]$ has diagonal elements equal to 1.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>uplo</code>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is packed in the array <code>ap</code> : If <code>uplo</code> = 'U', the array <code>ap</code> stores the upper triangular part of the matrix A . If <code>uplo</code> = 'L', the array <code>ap</code> stores the lower triangular part of the matrix A .
<code>n</code>	The order of matrix A ; $n \geq 0$.
<code>ap</code>	Array, size at least $\max(1, n(n+1)/2)$. The array <code>ap</code> contains the upper or the lower triangular part of the matrix A (as specified by <code>uplo</code>) in <i>packed storage</i> (see Matrix Storage Schemes).

Output Parameters

<code>s</code>	Array, size (n) . If <code>info</code> = 0, the array <code>s</code> contains the scale factors for A .
<code>scond</code>	If <code>info</code> = 0, <code>scond</code> contains the ratio of the smallest $s[i]$ to the largest $s[i]$.
<code>amax</code>	Absolute value of the largest element of the matrix A .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to SMLNUM or very close to BIGNUM, the matrix *A* should be scaled.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?pbequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive-definite band matrix and reduce its condition number.

Syntax

```
lapack_int LAPACKE_spbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const float* ab, lapack_int ldab, float* s, float* acond, float* amax );

lapack_int LAPACKE_dpbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const double* ab, lapack_int ldab, double* s, double* acond, double* amax );

lapack_int LAPACKE_cpbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const lapack_complex_float* ab, lapack_int ldab, float* s, float* acond, float* amax );

lapack_int LAPACKE_zpbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const lapack_complex_double* ab, lapack_int ldab, double* s, double* acond, double*
amax );
```

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite band matrix *A* and reduce its condition number (with respect to the two-norm). The output array *s* returns scale factors such that contains

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix *B* with elements $b_{ij}=s[i-1]*a_{ij}*s[j-1]$ has diagonal elements equal to 1. This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of <i>A</i> is stored in the array <i>ab</i>:</p> <p>If <i>uplo</i> = 'U', the array <i>ab</i> stores the upper triangular part of the matrix <i>A</i>.</p> <p>If <i>uplo</i> = 'L', the array <i>ab</i> stores the lower triangular part of the matrix <i>A</i>.</p>
<i>n</i>	The order of matrix <i>A</i> ; $n \geq 0$.
<i>kd</i>	The number of superdiagonals or subdiagonals in the matrix <i>A</i> ; $kd \geq 0$.
<i>ab</i>	<p>Array, size $\max(1, ldab*n)$.</p> <p>The array <i>ap</i> contains either the upper or the lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i>) in <i>band storage</i> (see Matrix Storage Schemes).</p>
<i>ldab</i>	The leading dimension of the array <i>ab</i> ; $ldab \geq kd + 1$.

Output Parameters

<i>s</i>	<p>Array, size (<i>n</i>).</p> <p>If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i>.</p>
<i>scond</i>	If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> [<i>i</i>] to the largest <i>s</i> [<i>i</i>].
<i>amax</i>	Absolute value of the largest element of the matrix <i>A</i> .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

Application Notes

If $scond \geq 0.1$ and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to SMLNUM or very close to BIGNUM, the matrix *A* should be scaled.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?syequb

Computes row and column scaling factors intended to equilibrate a symmetric indefinite matrix and reduce its condition number.

Syntax

```
lapack_int LAPACKKE_ssyequb( int matrix_layout, char uplo, lapack_int n, const float* a,
lapack_int lda, float* s, float* scond, float* amax );
```

```
lapack_int LAPACKKE_dsyequb( int matrix_layout, char uplo, lapack_int n, const double*
a, lapack_int lda, double* s, double* scond, double* amax );
```

```
lapack_int LAPACKKE_csyequb( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* a, lapack_int lda, float* s, float* scond, float* amax );
```

```
lapack_int LAPACKKE_zsyequb( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* a, lapack_int lda, double* s, double* scond, double* amax );
```

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric indefinite matrix A and reduce its condition number (with respect to the two-norm).

The array s contains the scale factors, $s[i-1] = 1/\sqrt{A(i,i)}$. These factors are chosen so that the scaled matrix B with elements $b_{i,j}=s[i-1]*a_{i,j}*s[j-1]$ has ones on the diagonal.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the array a stores the upper triangular part of the matrix A . If <i>uplo</i> = 'L', the array a stores the lower triangular part of the matrix A .
<i>n</i>	The order of the matrix A ; $n \geq 0$.
<i>a</i>	Array a : $\max(1, lda*n)$. Contains the n -by- n symmetric indefinite matrix A whose scaling factors are to be computed. Only the diagonal elements of A are referenced.
<i>lda</i>	The leading dimension of a ; $lda \geq \max(1, m)$.

Output Parameters

<i>s</i>	Array, size (<i>n</i>). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> [<i>i</i>] to the largest <i>s</i> [<i>i</i>]. If <i>scond</i> ≥ 0.1, and <i>amax</i> is neither too large nor too small, it is not worth scaling by <i>s</i> .
<i>amax</i>	Absolute value of the largest element of the matrix <i>A</i> . If <i>amax</i> is very close to SMLNUM or BIGNUM, the matrix should be scaled.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

?heequb

Computes row and column scaling factors intended to equilibrate a Hermitian indefinite matrix and reduce its condition number.

Syntax

```
lapack_int LAPACKE_cheequb( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* a, lapack_int lda, float* s, float* sconf, float* amax );

lapack_int LAPACKE_zheequb( int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* a, lapack_int lda, double* s, double* sconf, double* amax );
```

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate a Hermitian indefinite matrix *A* and reduce its condition number (with respect to the two-norm).

The array *s* contains the scale factors, $s[i-1] = 1/\sqrt{a_{i,i}}$. These factors are chosen so that the scaled matrix *B* with elements $b_{i,j}=s[i-1]*a_{i,j}*s[j-1]$ has ones on the diagonal.

This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
----------------------	---

<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is stored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix <i>A</i> . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ; $n \geq 0$.
<i>a</i>	Array <i>a</i> : size $\max(1, lda*n)$. Contains the <i>n</i> -by- <i>n</i> symmetric indefinite matrix <i>A</i> whose scaling factors are to be computed. Only the diagonal elements of <i>A</i> are referenced.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, m)$.

Output Parameters

<i>s</i>	Array, size (<i>n</i>). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> [<i>i</i>] to the largest <i>s</i> [<i>i</i>]. If <i>scond</i> ≥ 0.1 , and <i>amax</i> is neither too large nor too small, it is not worth scaling by <i>s</i> .
<i>amax</i>	Absolute value of the largest element of the matrix <i>A</i> . If <i>amax</i> is very close to SMLNUM or BIGNUM, the matrix should be scaled.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

See Also

[Error Analysis](#)

[Matrix Storage Schemes](#)

LAPACK Linear Equation Driver Routines

Table "Driver Routines for Solving Systems of Linear Equations" lists the LAPACK driver routines for solving systems of linear equations with real or complex matrices.

Driver Routines for Solving Systems of Linear Equations

Matrix type, storage scheme	Simple Driver	Expert Driver	Expert Driver using Extra-Precise Iterative Refinement
general	?gesv	?gesvx	?gesvxx
general band	?gbsv	?gbsvx	?gbsvxx

Matrix type, storage scheme	Simple Driver	Expert Driver	Expert Driver using Extra-Precise Iterative Refinement
general tridiagonal	?gtsv	?gtsvx	
diagonally dominant tridiagonal	?dtsvb		
symmetric/Hermitian positive-definite	?posv	?posvx	?posvxx
symmetric/Hermitian positive-definite, storage	?ppsv	?ppsvx	
symmetric/Hermitian positive-definite, band	?pbsv	?pbsvx	
symmetric/Hermitian positive-definite, tridiagonal	?ptsv	?ptsvx	
symmetric/Hermitian indefinite	?sysv/?hesv ?sysv_rook/?sysv_rk/?hesv_rk ?sysv_aa/?hesv_aa	?sysvx/?hesvx	?sysvxx/?hesvxx
symmetric/Hermitian indefinite, packed storage	?spsv/?hpsv	?spsvx/?hpsvx	
complex symmetric	?sysv ?sysv_rook	?sysvx	
complex symmetric, packed storage	?spsv	?spsvx	

In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex). In the description of [?gesv](#) and [?posv](#) routines, the ? sign stands for combined character codes ds and zc for the mixed precision subroutines.

[?gesv](#)

Computes the solution to the system of linear equations with a square coefficient matrix A and multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_sgesv (int matrix_layout , lapack_int n , lapack_int nrhs , float *
a , lapack_int lda , lapack_int * ipiv , float * b , lapack_int ldb );

lapack_int LAPACKE_dgesv (int matrix_layout , lapack_int n , lapack_int nrhs , double *
a , lapack_int lda , lapack_int * ipiv , double * b , lapack_int ldb );

lapack_int LAPACKE_cgesv (int matrix_layout , lapack_int n , lapack_int nrhs ,
lapack_complex_float * a , lapack_int lda , lapack_int * ipiv , lapack_complex_float *
b , lapack_int ldb );
```



```

lapack_int LAPACKE_zgesv (int matrix_layout , lapack_int n , lapack_int nrhs ,
lapack_complex_double * a , lapack_int lda , lapack_int * ipiv , lapack_complex_double
* b , lapack_int ldb );

lapack_int LAPACKE_dsgeev (int matrix_layout, lapack_int n, lapack_int nrhs, double *
a, lapack_int lda, lapack_int * ipiv, double * b, lapack_int ldb, double * x, lapack_int
ldx, lapack_int * iter);

lapack_int LAPACKE_zcgesv (int matrix_layout, lapack_int n, lapack_int nrhs,
lapack_complex_double * a, lapack_int lda, lapack_int * ipiv, lapack_complex_double *
b, lapack_int ldb, lapack_complex_double * x, lapack_int ldx, lapack_int * iter);

```

Include Files

- mkl.h

Description

The routine solves for X the system of linear equations $A \cdot X = B$, where A is an n -by- n matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The LU decomposition with partial pivoting and row interchanges is used to factor A as $A = P \cdot L \cdot U$, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations $A \cdot X = B$.

The `dsgeev` and `zcgesv` are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (`dsgeev`) or single complex precision (`zcgesv`) and use this factorization within an iterative refinement procedure to produce a solution with double precision (`dsgeev`) / double complex precision (`zcgesv`) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.

The iterative refinement is not going to be a winning strategy if the ratio single precision performance over double precision performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to `ilaenv` in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if

```
iter > itermax
```

or for all the right-hand sides:

```
rnrm < sqrt(n)*xnrm*anrm*eps*bwdmax
```

where

- `iter` is the number of the current iteration in the iterative refinement process
- `rnrm` is the infinity-norm of the residual
- `xnrm` is the infinity-norm of the solution
- `anrm` is the infinity-operator-norm of the matrix A
- `eps` is the machine epsilon returned by `dlamch` ('Epsilon').

The values `itermax` and `bwdmax` are fixed to 30 and 1.0d+00 respectively.

Input Parameters

`matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

<i>n</i>	The number of linear equations, that is, the order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, that is, the number of columns of the matrix <i>B</i> ; $nrhs \geq 0$.
<i>a</i>	The array <i>a</i> (size $\max(1, lda*n)$) contains the <i>n</i> -by- <i>n</i> coefficient matrix <i>A</i> .
<i>b</i>	The array <i>b</i> of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the <i>n</i> -by- <i>nrhs</i> matrix of right hand side matrix <i>B</i> .
<i>lda</i>	The leading dimension of the array <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ldx</i>	The leading dimension of the array <i>x</i> ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

<i>a</i>	<p>Overwritten by the factors <i>L</i> and <i>U</i> from the factorization of $A = P*L*U$; the unit diagonal elements of <i>L</i> are not stored.</p> <p>If iterative refinement has been successfully used (<i>info</i>= 0 and <i>iter</i>≥ 0), then <i>A</i> is unchanged.</p> <p>If double precision factorization has been used (<i>info</i>= 0 and <i>iter</i> < 0), then the array <i>A</i> contains the factors <i>L</i> and <i>U</i> from the factorization $A = P*L*U$; the unit diagonal elements of <i>L</i> are not stored.</p>
<i>b</i>	<p>Overwritten by the solution matrix <i>X</i> for <i>dgesv</i>, <i>sgesv</i>, <i>zgesv</i>, <i>zgesv</i>. Unchanged for <i>dsgesv</i> and <i>zcgesv</i>.</p>
<i>ipiv</i>	<p>Array, size at least $\max(1, n)$. The pivot indices that define the permutation matrix <i>P</i>; row <i>i</i> of the matrix was interchanged with row <i>ipiv</i>[<i>i</i>-1]. Corresponds to the single precision factorization (if <i>info</i>= 0 and <i>iter</i>≥ 0) or the double precision factorization (if <i>info</i>= 0 and <i>iter</i> < 0).</p>
<i>x</i>	<p>Array, size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout. If <i>info</i> = 0, contains the <i>n</i>-by-<i>nrhs</i> solution matrix <i>X</i>.</p>
<i>iter</i>	<p>If <i>iter</i> < 0: iterative refinement has failed, double precision factorization has been performed</p> <ul style="list-style-type: none"> • If <i>iter</i> = -1: the routine fell back to full precision for implementation- or machine-specific reason • If <i>iter</i> = -2: narrowing the precision induced an overflow, the routine fell back to full precision • If <i>iter</i> = -3: failure of <i>sgetrf</i> for <i>dsgesv</i>, or <i>cgetrf</i> for <i>zcgesv</i> • If <i>iter</i> = -31: stop the iterative refinement after the 30th iteration.

If *iter* > 0: iterative refinement has been successfully used. Returns the number of iterations.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, $U_{i,i}$ (computed in double precision for mixed precision subroutines) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

See Also

[dlamch](#)

[sgetrf](#)

Matrix Storage Schemes

?gesvx

Computes the solution to the system of linear equations with a square coefficient matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
lapack_int LAPACKESgesvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, lapack_int* ipiv,
char* equed, float* r, float* c, float* b, lapack_int ldb, float* x, lapack_int ldx,
float* rcond, float* ferr, float* berr, float* rpivot );
```

```
lapack_int LAPACKEDgesvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, lapack_int*
ipiv, char* equed, double* r, double* c, double* b, lapack_int ldb, double* x,
lapack_int ldx, double* rcond, double* ferr, double* berr, double* rpivot );
```

```
lapack_int LAPACKESgesvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, float* r, float* c,
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* rcond, float* ferr, float* berr, float* rpivot );
```

```
lapack_int LAPACKESgesvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, double* r, double* c,
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* rcond, double* ferr, double* berr, double* rpivot );
```

Include Files

- `mk1.h`

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where *A* is an *n*-by-*n* matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gesvx` performs the following steps:

1. If `fact = 'E'`, real scaling factors r and c are computed to equilibrate the system:

$$\text{trans} = 'N': \text{diag}(r) * A * \text{diag}(c) * \text{inv}(\text{diag}(c)) * X = \text{diag}(r) * B$$

$$\text{trans} = 'T': (\text{diag}(r) * A * \text{diag}(c))^T * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$$

$$\text{trans} = 'C': (\text{diag}(r) * A * \text{diag}(c))^H * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(r) * A * \text{diag}(c)$ and B by $\text{diag}(r) * B$ (if `trans='N'`) or $\text{diag}(c) * B$ (if `trans = 'T' or 'C'`).

2. If `fact = 'N' or 'E'`, the LU decomposition is used to factor the matrix A (after equilibration if `fact = 'E'`) as $A = P * L * U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n + 1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(c)$ (if `trans = 'N'`) or $\text{diag}(r)$ (if `trans = 'T' or 'C'`) so that it solves the original system before equilibration.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>fact</code>	<p>Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If <code>fact = 'F'</code>: on entry, <code>af</code> and <code>ipiv</code> contain the factored form of A. If <code>equed</code> is not 'N', the matrix A has been equilibrated with scaling factors given by r and c. a, <code>af</code>, and <code>ipiv</code> are not modified.</p> <p>If <code>fact = 'N'</code>, the matrix A will be copied to <code>af</code> and factored.</p> <p>If <code>fact = 'E'</code>, the matrix A will be equilibrated if necessary, then copied to <code>af</code> and factored.</p>
<code>trans</code>	<p>Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <code>trans = 'N'</code>, the system has the form $A * X = B$ (No transpose).</p> <p>If <code>trans = 'T'</code>, the system has the form $A^T * X = B$ (Transpose).</p> <p>If <code>trans = 'C'</code>, the system has the form $A^H * X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).</p>
<code>n</code>	The number of linear equations; the order of the matrix A ; $n \geq 0$.

<i>nrhs</i>	The number of right hand sides; the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.
<i>a</i>	The array <i>a</i> (size $\max(1, lda*n)$) contains the matrix <i>A</i> . If <i>fact</i> = 'F' and <i>equed</i> is not 'N', then <i>A</i> must have been equilibrated by the scaling factors in <i>r</i> and/or <i>c</i> .
<i>af</i>	The array <i>afaf</i> (size $\max(1, ldaf*n)$) is an input argument if <i>fact</i> = 'F'. It contains the factored form of the matrix <i>A</i> , that is, the factors <i>L</i> and <i>U</i> from the factorization $A = P*L*U$ as computed by ?getrf . If <i>equed</i> is not 'N', then <i>af</i> is the factored form of the equilibrated matrix <i>A</i> .
<i>b</i>	The array <i>b</i> of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. It contains the pivot indices from the factorization $A = P*L*U$ as computed by ?getrf ; row <i>i</i> of the matrix was interchanged with row <i>ipiv</i> [<i>i</i> -1].
<i>equed</i>	<p>Must be 'N', 'R', 'C', or 'B'.</p> <p><i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done:</p> <p>If <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N').</p> <p>If <i>equed</i> = 'R', row equilibration was done, that is, <i>A</i> has been premultiplied by <i>diag</i>(<i>r</i>).</p> <p>If <i>equed</i> = 'C', column equilibration was done, that is, <i>A</i> has been postmultiplied by <i>diag</i>(<i>c</i>).</p> <p>If <i>equed</i> = 'B', both row and column equilibration was done, that is, <i>A</i> has been replaced by <i>diag</i>(<i>r</i>)*<i>A</i>*<i>diag</i>(<i>c</i>).</p>
<i>r, c</i>	<p>Arrays: <i>r</i> (size <i>n</i>), <i>c</i> (size <i>n</i>). The array <i>r</i> contains the row scale factors for <i>A</i>, and the array <i>c</i> contains the column scale factors for <i>A</i>. These arrays are input arguments if <i>fact</i> = 'F' only; otherwise they are output arguments.</p> <p>If <i>equed</i> = 'R' or 'B', <i>A</i> is multiplied on the left by <i>diag</i>(<i>r</i>); if <i>equed</i> = 'N' or 'C', <i>r</i> is not accessed.</p> <p>If <i>fact</i> = 'F' and <i>equed</i> = 'R' or 'B', each element of <i>r</i> must be positive.</p> <p>If <i>equed</i> = 'C' or 'B', <i>A</i> is multiplied on the right by <i>diag</i>(<i>c</i>); if <i>equed</i> = 'N' or 'R', <i>c</i> is not accessed.</p>

If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.

ldx

The leading dimension of the output array *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

x

Array, size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the *equilibrated* system is:

$diag(C)^{-1} * X$, if *trans* = 'N' and *equed* = 'C' or 'B';
 $diag(R)^{-1} * X$, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'. The second dimension of *x* must be at least $\max(1, nrhs)$.

a

Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'. If *equed* ≠ 'N', *A* is scaled on exit as follows:

equed = 'R': $A = diag(R) * A$
equed = 'C': $A = A * diag(c)$
equed = 'B': $A = diag(R) * A * diag(c)$.

af

If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization $A = PLU$ of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b

Overwritten by $diag(r) * B$ if *trans* = 'N' and *equed* = 'R' or 'B';
 overwritten by $diag(c) * B$ if *trans* = 'T' or 'C' and *equed* = 'C' or 'B';
 not changed if *equed* = 'N'.

r, c

These arrays are output arguments if *fact* ≠ 'F'. See the description of *r, c* in *Input Arguments* section.

rcond

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector x_j (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to x_j , *ferr*[*j*-1] is an estimated upper bound for the magnitude of the largest element in $(x_j - x_{true})$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

<i>berr</i>	Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of A or B that makes x_j an exact solution.
<i>ipiv</i>	If <i>fact</i> = 'N' or 'E', then <i>ipiv</i> is an output argument and on exit contains the pivot indices from the factorization $A = P*L*U$ of the original matrix A (if <i>fact</i> = 'N') or of the equilibrated matrix A (if <i>fact</i> = 'E').
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>rpivot</i>	On exit, <i>rpivot</i> contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If *rpivot* is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution x , condition estimator *rcond*, and forward error bound *ferr* could be unreliable. If factorization fails with $0 < info \leq n$, then *rpivot* contains the reciprocal pivot growth factor for the leading *info* columns of A .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = $n + 1$, then U is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

See Also

Matrix Storage Schemes

?gesvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a square coefficient matrix A and multiple right-hand sides

Syntax

```
lapack_int LAPACKE_sgesvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, lapack_int* ipiv,
char* equed, float* r, float* c, float* b, lapack_int ldb, float* x, lapack_int ldx,
float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm,
float* err_bnds_comp, lapack_int nparams, const float* params );
```

```
lapack_int LAPACKE_dgesvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, lapack_int*
ipiv, char* equed, double* r, double* c, double* b, lapack_int ldb, double* x,
lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds,
double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double*
params );
```

```
lapack_int LAPACKE_cgesvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, float* r, float* c,
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm,
float* err_bnds_comp, lapack_int nparams, const float* params );
```

```
lapack_int LAPACKE_zgesvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, double* r, double* c,
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds, double*
err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );
```

Include Files

- mkl.h

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is an n -by- n matrix, the columns of the matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine `?gesvxx` performs the following steps:

1. If *fact* = 'E', scaling factors r and c are computed to equilibrate the system:

$$\text{trans} = \text{'N'}: \text{diag}(r) * A * \text{diag}(c) * \text{inv}(\text{diag}(c)) * X = \text{diag}(r) * B$$

$$\text{trans} = \text{'T'}: (\text{diag}(r) * A * \text{diag}(c))^T * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$$

$$\text{trans} = \text{'C'}: (\text{diag}(r) * A * \text{diag}(c))^H * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(r) * A * \text{diag}(c)$ and B by $\text{diag}(r) * B$ (if *trans*='N') or $\text{diag}(c) * B$ (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as $A = P * L * U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.

3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless is set to zero, the routine applies iterative refinement to improve the computed solution matrix and calculate error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $diag(c)$ (if $trans = 'N'$) or $diag(r)$ (if $trans = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>fact</i>	<p>Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If $fact = 'F'$, on entry, af and $ipiv$ contain the factored form of A. If $equed$ is not 'N', the matrix A has been equilibrated with scaling factors given by r and c. Parameters a, af, and $ipiv$ are not modified.</p> <p>If $fact = 'N'$, the matrix A will be copied to af and factored.</p> <p>If $fact = 'E'$, the matrix A will be equilibrated, if necessary, copied to af and factored.</p>
<i>trans</i>	<p>Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If $trans = 'N'$, the system has the form $A^*X = B$ (No transpose).</p> <p>If $trans = 'T'$, the system has the form $A^T X = B$ (Transpose).</p> <p>If $trans = 'C'$, the system has the form $A^H X = B$ (Conjugate Transpose = Transpose for real flavors, Conjugate Transpose for complex flavors).</p>
<i>n</i>	The number of linear equations; the order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<i>a</i> , <i>af</i> , <i>b</i>	<p>Arrays: a(size $\max(lda*n)$), af(size $\max(ldaf*n)$), b(size $\max(1, ldb*nrhs)$) for column major layout and $\max(1, ldb*n)$ for row major layout).</p> <p>The array a contains the matrix A. If $fact = 'F'$ and $equed$ is not 'N', then A must have been equilibrated by the scaling factors in r and/or c.</p>

The array *af* is an input argument if *fact* = 'F'. It contains the factored form of the matrix *A*, that is, the factors *L* and *U* from the factorization $A = P * L * U$ as computed by `?getrf`. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

lda

The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf

The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ipiv

Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization $A = P * L * U$ as computed by `?getrf`; row *i* of the matrix was interchanged with row *ipiv*[*i*-1].

equed

Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag*(*r*).

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag*(*c*).

If *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*r*) * *A* * *diag*(*c*).

r, c

Arrays: *r* (size *n*), *c* (size *n*). The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.

If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed.

If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed.

If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.

Each element of *r* or *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

The leading dimension of the array *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

<code>ldx</code>	The leading dimension of the output array <code>x</code> ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.									
<code>n_err_bnds</code>	Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <code>err_bnds_norm</code> and <code>err_bnds_comp</code> descriptions in <i>Output Arguments</i> section below.									
<code>nparams</code>	Specifies the number of parameters set in <code>params</code> . If ≤ 0 , the <code>params</code> array is never referenced and default values are used.									
<code>params</code>	<p>Array, size $\max(1, nparams)$. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to <code>nparams</code> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <code>nparams = 0</code>, which prevents the source code from accessing the <code>params</code> argument.</p> <p><code>params[0]</code> : Whether to perform iterative refinement or not. Default: 1.0</p> <table><tr><td>=0.0</td><td>No refinement is performed and no error bounds are computed.</td></tr><tr><td>=1.0</td><td>Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.</td></tr></table> <p>(Other values are reserved for future use.)</p> <p><code>params[1]</code> : Maximum number of residual computations allowed for refinement.</p> <table><tr><td>Default</td><td>10.0</td></tr><tr><td>Aggressive</td><td>Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i>. If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy.</td></tr></table> <p><code>params[2]</code> : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).</p>		=0.0	No refinement is performed and no error bounds are computed.	=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.	Default	10.0	Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy.
=0.0	No refinement is performed and no error bounds are computed.									
=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.									
Default	10.0									
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy.									

Output Parameters

<code>x</code>	<p>Array, size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout.</p> <p>If <code>info = 0</code>, the array <code>x</code> contains the solution n-by-<code>nrhs</code> matrix X to the <i>original</i> system of equations. Note that A and B are modified on exit if <code>equet</code> \neq 'N', and the solution to the <i>equilibrated</i> system is:</p>
----------------	--

$\text{inv}(\text{diag}(c)) * X$, if $\text{trans} = 'N'$ and $\text{equed} = 'C'$ or $'B'$; or
 $\text{inv}(\text{diag}(r)) * X$, if $\text{trans} = 'T'$ or $'C'$ and $\text{equed} = 'R'$ or $'B'$.

a

Array *a* is not modified on exit if $\text{fact} = 'F'$ or $'N'$, or if $\text{fact} = 'E'$ and $\text{equed} = 'N'$.

If $\text{equed} \neq 'N'$, *A* is scaled on exit as follows:

$\text{equed} = 'R': A = \text{diag}(r) * A$

$\text{equed} = 'C': A = A * \text{diag}(c)$

$\text{equed} = 'B': A = \text{diag}(r) * A * \text{diag}(c)$.

af

If $\text{fact} = 'N'$ or $'E'$, then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization $A = PLU$ of the original matrix *A* (if $\text{fact} = 'N'$) or of the equilibrated matrix *A* (if $\text{fact} = 'E'$). See the description of *a* for the form of the equilibrated matrix.

b

Overwritten by $\text{diag}(r) * B$ if $\text{trans} = 'N'$ and $\text{equed} = 'R'$ or $'B'$;

overwritten by $\text{trans} = 'T'$ or $'C'$ and $\text{equed} = 'C'$ or $'B'$;

not changed if $\text{equed} = 'N'$.

r, c

These arrays are output arguments if $\text{fact} \neq 'F'$. Each element of these arrays is a power of the radix. See the description of *r, c* in *Input Arguments* section.

rcond

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if $\text{rcond} = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < \text{info} \leq n$, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*. In *?gesvx*, this quantity is returned in *rpivot*.

berr

Array, size at least $\max(1, \text{nrhs})$. Contains the componentwise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of *A* or *B* that makes x_j an exact solution.

err_bnds_norm

Array of size $\text{nrhs} * n_err_bnds$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

`err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3` Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z=s*a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error `err` is stored in `err_bnds_norm[(err-1)*nrhs + i - 1]`.

`err_bnds_comp`

Array of size `nrhs*n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params*[2] = 0.0), then *err_bnds_comp* is not accessed.

<i>err</i> =1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
<i>err</i> =2	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<i>err</i> =3	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix <i>Z</i> are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where *x* is the solution for the current right-hand side and *s* scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of *z* are approximately 1.

The information for right-hand side *i*, where $1 \leq i \leq \text{nrhs}$, and type of error *err* is stored in *err_bnds_comp*[(*err*-1)**nrhs* + *i* - 1].

<i>ipiv</i>	If <i>fact</i> = 'N' or 'E', then <i>ipiv</i> is an output argument and on exit contains the pivot indices from the factorization $A = P * L * U$ of the original matrix <i>A</i> (if <i>fact</i> = 'N') or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E').
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>params</i>	If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If $info = -i$, parameter i had an illegal value.

If $0 < info \leq n$: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $params[2] = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that for column major layout $err_bnds_norm[j - 1] = 0.0$ or $err_bnds_comp[j - 1] = 0.0$; or for row major layout $err_bnds_norm[(j - 1)*n_err_bnds] = 0.0$ or $err_bnds_comp[(j - 1)*n_err_bnds] = 0.0$). See the definition of err_bnds_norm and err_bnds_comp for $err = 1$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

See Also

Matrix Storage Schemes

?gbsv

Computes the solution to the system of linear equations with a band coefficient matrix A and multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_sgbsv (int matrix_layout , lapack_int n , lapack_int kl , lapack_int
ku , lapack_int nrhs , float * ab , lapack_int ldab , lapack_int * ipiv , float * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_dgbsv (int matrix_layout , lapack_int n , lapack_int kl , lapack_int
ku , lapack_int nrhs , double * ab , lapack_int ldab , lapack_int * ipiv , double * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_cgbsv (int matrix_layout , lapack_int n , lapack_int kl , lapack_int
ku , lapack_int nrhs , lapack_complex_float * ab , lapack_int ldab , lapack_int *
ipiv , lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_zgbsv (int matrix_layout , lapack_int n , lapack_int kl , lapack_int
ku , lapack_int nrhs , lapack_complex_double * ab , lapack_int ldab , lapack_int *
ipiv , lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for X the real or complex system of linear equations $A * X = B$, where A is an n -by- n band matrix with kl subdiagonals and ku superdiagonals, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The LU decomposition with partial pivoting and row interchanges is used to factor A as $A = L * U$, where L is a product of permutation and unit lower triangular matrices with kl subdiagonals, and U is upper triangular with $kl+ku$ superdiagonals. The factored form of A is then used to solve the system of equations $A * X = B$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>n</i>	The order of <i>A</i> . The number of rows in <i>B</i> ; $n \geq 0$.
<i>kl</i>	The number of subdiagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>nrhs</i>	The number of right-hand sides. The number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>ab, b</i>	Arrays: <i>ab</i> (size $\max(1, ldab*n)$), <i>b</i> of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout. The array <i>ab</i> contains the matrix <i>A</i> in band storage (see Matrix Storage Schemes). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
<i>ldab</i>	The leading dimension of the array <i>ab</i> . ($ldab \geq 2kl + ku + 1$)
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>ab</i>	Overwritten by <i>L</i> and <i>U</i> . <i>U</i> is stored as an upper triangular band matrix with $kl + ku$ superdiagonals and <i>L</i> is stored as a lower triangular band matrix with <i>kl</i> subdiagonals. See Matrix Storage Schemes .
<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	Array, size at least $\max(1, n)$. The pivot indices: row <i>i</i> was interchanged with row <i>ipiv</i> [<i>i</i> -1].

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, $U_{i,i}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

See Also

[Matrix Storage Schemes](#)

?gbsvx

*Computes the solution to the real or complex system of linear equations with a band coefficient matrix *A* and multiple right-hand sides, and provides error bounds on the solution.*

Syntax

```
lapack_int LAPACKES_gbsvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, float* ab, lapack_int ldab, float* afb,
lapack_int ldafb, lapack_int* ipiv, char* equed, float* r, float* c, float* b,
lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr, float*
rpivot );
```

```
lapack_int LAPACKES_dgbvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, double* ab, lapack_int ldab, double* afb,
lapack_int ldafb, lapack_int* ipiv, char* equed, double* r, double* c, double* b,
lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr,
double* rpivot );
```

```
lapack_int LAPACKES_cgbvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_float* ab, lapack_int
ldab, lapack_complex_float* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
float* r, float* c, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* ferr, float* berr, float* rpivot );
```

```
lapack_int LAPACKES_zgbvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_double* ab, lapack_int
ldab, lapack_complex_double* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
double* r, double* c, lapack_complex_double* b, lapack_int ldb, lapack_complex_double*
x, lapack_int ldx, double* rcond, double* ferr, double* berr, double* rpivot );
```

Include Files

- mkl.h

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, $A^T \cdot X = B$, or $A^H \cdot X = B$, where A is a band matrix of order n with kl subdiagonals and ku superdiagonals, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gbsvx` performs the following steps:

1. If *fact* = 'E', real scaling factors r and c are computed to equilibrate the system:

trans = 'N': $\text{diag}(r) * A * \text{diag}(c) * \text{inv}(\text{diag}(c)) * X = \text{diag}(r) * B$

trans = 'T': $(\text{diag}(r) * A * \text{diag}(c))^T * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$

trans = 'C': $(\text{diag}(r) * A * \text{diag}(c))^H * \text{inv}(\text{diag}(r)) * X = \text{diag}(c) * B$

Whether the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(r) * A * \text{diag}(c)$ and B by $\text{diag}(r) * B$ (if *trans* = 'N') or $\text{diag}(c) * B$ (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as $A = L \cdot U$, where L is a product of permutation and unit lower triangular matrices with kl subdiagonals, and U is upper triangular with $kl + ku$ superdiagonals.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with *info* = i . Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, *info* = $n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(c)$ (if $\text{trans} = 'N'$) or $\text{diag}(r)$ (if $\text{trans} = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>fact</code>	<p>Must be 'F', 'N', or 'E'.</p> <p>Specifies whether the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If $\text{fact} = 'F'$: on entry, afb and ipiv contain the factored form of A. If equed is not 'N', the matrix A is equilibrated with scaling factors given by r and c. ab, afb, and ipiv are not modified.</p> <p>If $\text{fact} = 'N'$, the matrix A will be copied to afb and factored.</p> <p>If $\text{fact} = 'E'$, the matrix A will be equilibrated if necessary, then copied to afb and factored.</p>
<code>trans</code>	<p>Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If $\text{trans} = 'N'$, the system has the form $A * X = B$ (No transpose).</p> <p>If $\text{trans} = 'T'$, the system has the form $A^T * X = B$ (Transpose).</p> <p>If $\text{trans} = 'C'$, the system has the form $A^H * X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).</p>
<code>n</code>	The number of linear equations, the order of the matrix A ; $n \geq 0$.
<code>kl</code>	The number of subdiagonals within the band of A ; $kl \geq 0$.
<code>ku</code>	The number of superdiagonals within the band of A ; $ku \geq 0$.
<code>nrhs</code>	The number of right hand sides, the number of columns of the matrices B and X ; $\text{nrhs} \geq 0$.
<code>ab, afb, b</code>	<p>Arrays: ab ($\max(\text{ldab} * n)$), afb ($\max(\text{ldaafb} * n)$), b ($\max(1, \text{ldb} * \text{nrhs})$) for column major layout and $\max(1, \text{ldb} * n)$ for row major layout).</p> <p>The array ab contains the matrix A in band storage (see Matrix Storage Schemes). If $\text{fact} = 'F'$ and equed is not 'N', then A must have been equilibrated by the scaling factors in r and/or c.</p> <p>The array afb is an input argument if $\text{fact} = 'F'$. It contains the factored form of the matrix A, that is, the factors L and U from the factorization $A = P * L * U$ as computed by <code>?gbtrf</code>. U is stored as an upper triangular band matrix with $kl + ku$ superdiagonals. L is stored as lower triangular band matrix with kl subdiagonals. If equed is not 'N', then afb is the factored form of the equilibrated matrix A.</p>

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldab

The leading dimension of *ab*; $ldab \geq kl + ku + 1$.

ldaafb

The leading dimension of *afb*; $ldaafb \geq 2 * kl + ku + 1$.

ldb

The leading dimension of *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ipiv

Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains the pivot indices from the factorization $A = P * L * U$ as computed by `?gbtrf`; row *i* of the matrix was interchanged with row *ipiv*[*i*-1].

equed

Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag*(*r*).

If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag*(*c*).

if *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag*(*r*) * *A* * *diag*(*c*).

r, c

Arrays: *r* (size *n*), *c* (size *n*).

The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.

If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag*(*r*); if *equed* = 'N' or 'C', *r* is not accessed.

If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.

If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag*(*c*); if *equed* = 'N' or 'R', *c* is not accessed.

If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.

ldx

The leading dimension of the output array *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

x

Array, size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout.

If $info = 0$ or $info = n+1$, the array x contains the solution matrix X to the *original* system of equations. Note that A and B are modified on exit if $equed \neq 'N'$, and the solution to the *equilibrated* system is:
 $inv(diag(c)) * X$, if $trans = 'N'$ and $equed = 'C'$ or $'B'$;
 $inv(diag(r)) * X$, if $trans = 'T'$ or $'C'$ and $equed = 'R'$ or $'B'$.

ab

Array *ab* is not modified on exit if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$.

If $equed \neq 'N'$, A is scaled on exit as follows:

$equed = 'R'$: $A = diag(r) * A$

$equed = 'C'$: $A = A * diag(c)$

$equed = 'B'$: $A = diag(r) * A * diag(c)$.

afb

If $fact = 'N'$ or $'E'$, then *afb* is an output argument and on exit returns details of the *LU* factorization of the original matrix A (if $fact = 'N'$) or of the equilibrated matrix A (if $fact = 'E'$). See the description of *ab* for the form of the equilibrated matrix.

b

Overwritten by $diag(r) * b$ if $trans = 'N'$ and $equed = 'R'$ or $'B'$;

overwritten by $diag(c) * b$ if $trans = 'T'$ or $'C'$ and $equed = 'C'$ or $'B'$;

not changed if $equed = 'N'$.

r, c

These arrays are output arguments if $fact \neq 'F'$. See the description of *r, c* in *Input Arguments* section.

rcond

An estimate of the reciprocal condition number of the matrix A after equilibration (if done).

If *rcond* is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr

Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector x_j (the j -th column of the solution matrix X). If x_{true} is the true solution corresponding to x_j , $ferr[j-1]$ is an estimated upper bound for the magnitude of the largest element in $(x_j - x_{true})$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of A or B that makes x_j an exact solution.

ipiv

If $fact = 'N'$ or $'E'$, then *ipiv* is an output argument and on exit contains the pivot indices from the factorization $A = L * U$ of the original matrix A (if $fact = 'N'$) or of the equilibrated matrix A (if $fact = 'E'$).

equed If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

rpivot On exit, *rpivot* contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If *rpivot* is much less than 1, then the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *x*, condition estimator *rcond*, and forward error bound *ferr* could be unreliable. If factorization fails with $0 < \text{info} \leq n$, then *rpivot* contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, and $i \leq n$, then $U_{i,i}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and $i = n+1$, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

See Also

Matrix Storage Schemes

?gbsvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a banded coefficient matrix A and multiple right-hand sides

Syntax

```
lapack_int LAPACKE_sgbvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, float* ab, lapack_int ldab, float* afb,
lapack_int ldafb, lapack_int* ipiv, char* equed, float* r, float* c, float* b,
lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* rpvgrw, float* berr,
lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams,
const float* params );
```

```
lapack_int LAPACKE_dgbvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, double* ab, lapack_int ldab, double* afb,
lapack_int ldafb, lapack_int* ipiv, char* equed, double* r, double* c, double* b,
lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr,
lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int
nparams, const double* params );
```

```
lapack_int LAPACKE_cgbvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_float* ab, lapack_int
ldab, lapack_complex_float* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
```

```
float* r, float* c, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float*
err_bnds_norm, float* err_bnds_comp, lapack_int nparams, const float* params );

lapack_int LAPACKE_zgbsvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_double* ab, lapack_int
ldab, lapack_complex_double* afb, lapack_int ldaafb, lapack_int* ipiv, char* equed,
double* r, double* c, lapack_complex_double* b, lapack_int ldb, lapack_complex_double*
x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds,
double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double*
params );
```

Include Files

- mkl.h

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, $A^T \cdot X = B$, or $A^H \cdot X = B$, where A is an n -by- n banded matrix, the columns of the matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine `?gbsvxx` performs the following steps:

1. If *fact* = 'E', scaling factors *r* and *c* are computed to equilibrate the system:

```
trans = 'N': diag(r)*A*diag(c)*inv(diag(c))*X = diag(r)*B
trans = 'T': (diag(r)*A*diag(c))^T*inv(diag(r))*X = diag(c)*B
trans = 'C': (diag(r)*A*diag(c))^H*inv(diag(r))*X = diag(c)*B
```

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(r) \cdot A \cdot \text{diag}(c)$ and B by $\text{diag}(r) \cdot B$ (if *trans*='N') or $\text{diag}(c) \cdot B$ (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as $A = P \cdot L \cdot U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with *info* = *i*. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless *params*[0] is set to zero, the routine applies iterative refinement to improve the computed solution matrix and calculate error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(c)$ (if *trans* = 'N') or $\text{diag}(r)$ (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

<i>matrix_layout</i>	Specifies whether two-dimensional array storage is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>fact</i>	<p>Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix <i>A</i> is supplied on entry, and if not, whether the matrix <i>A</i> should be equilibrated before it is factored.</p> <p>If <i>fact</i> = 'F', on entry, <i>afb</i> and <i>ipiv</i> contain the factored form of <i>A</i>. If <i>equed</i> is not 'N', the matrix <i>A</i> has been equilibrated with scaling factors given by <i>r</i> and <i>c</i>. Parameters <i>ab</i>, <i>afb</i>, and <i>ipiv</i> are not modified.</p> <p>If <i>fact</i> = 'N', the matrix <i>A</i> will be copied to <i>afb</i> and factored.</p> <p>If <i>fact</i> = 'E', the matrix <i>A</i> will be equilibrated, if necessary, copied to <i>afb</i> and factored.</p>
<i>trans</i>	<p>Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <i>trans</i> = 'N', the system has the form $A^*X = B$ (No transpose).</p> <p>If <i>trans</i> = 'T', the system has the form $A^T X = B$ (Transpose).</p> <p>If <i>trans</i> = 'C', the system has the form $A^H X = B$ (Conjugate Transpose = Transpose for real flavors, Conjugate Transpose for complex flavors).</p>
<i>n</i>	The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<i>kl</i>	The number of subdiagonals within the band of <i>A</i> ; $kl \geq 0$.
<i>ku</i>	The number of superdiagonals within the band of <i>A</i> ; $ku \geq 0$.
<i>nrhs</i>	The number of right-hand sides; the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.
<i>ab</i> , <i>afb</i> , <i>b</i>	<p>Arrays: <i>ab</i> (max(<i>ldab</i>*<i>n</i>)), <i>afb</i> (max(<i>ldafb</i>*<i>n</i>)), <i>b</i>(max(1, <i>ldb</i>*<i>nrhs</i>)) for column major layout and max(1, <i>ldb</i>*<i>n</i>) for row major layout).</p> <p>The array <i>ab</i> contains the matrix <i>A</i> in band storage.</p> <p>If <i>fact</i> = 'F' and <i>equed</i> is not 'N', then <i>AB</i> must have been equilibrated by the scaling factors in <i>r</i> and/or <i>c</i>.</p> <p>The array <i>afb</i> is an input argument if <i>fact</i> = 'F'. It contains the factored form of the banded matrix <i>A</i>, that is, the factors <i>L</i> and <i>U</i> from the factorization $A = P^*L^*U$ as computed by <code>?gbtrf</code>. <i>U</i> is stored as an upper triangular banded matrix with <i>kl</i> + <i>ku</i> superdiagonals. <i>L</i> is stored as lower triangular band matrix with <i>kl</i> subdiagonals. If <i>equed</i> is not 'N', then <i>afb</i> is the factored form of the equilibrated matrix <i>A</i>.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.</p>
<i>ldab</i>	The leading dimension of the array <i>ab</i> ; $ldab \geq kl + ku + 1$.

<i>lda_fb</i>	The leading dimension of the array <i>afb</i> ; $lda_fb \geq 2 * kl + ku + 1$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. It contains the pivot indices from the factorization $A = P * L * U$ as computed by <code>?gbtrf</code> ; row <i>i</i> of the matrix was interchanged with row <i>ipiv</i> [<i>i</i> -1].
<i>equed</i>	<p>Must be 'N', 'R', 'C', or 'B'.</p> <p><i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done:</p> <p>If <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N').</p> <p>If <i>equed</i> = 'R', row equilibration was done, that is, <i>A</i> has been premultiplied by <i>diag</i>(<i>r</i>).</p> <p>If <i>equed</i> = 'C', column equilibration was done, that is, <i>A</i> has been postmultiplied by <i>diag</i>(<i>c</i>).</p> <p>If <i>equed</i> = 'B', both row and column equilibration was done, that is, <i>A</i> has been replaced by <i>diag</i>(<i>r</i>) * <i>A</i> * <i>diag</i>(<i>c</i>).</p>
<i>r, c</i>	<p>Arrays: <i>r</i> (size <i>n</i>), <i>c</i> (size <i>n</i>). The array <i>r</i> contains the row scale factors for <i>A</i>, and the array <i>c</i> contains the column scale factors for <i>A</i>. These arrays are input arguments if <i>fact</i> = 'F' only; otherwise they are output arguments.</p> <p>If <i>equed</i> = 'R' or 'B', <i>A</i> is multiplied on the left by <i>diag</i>(<i>r</i>); if <i>equed</i> = 'N' or 'C', <i>r</i> is not accessed.</p> <p>If <i>fact</i> = 'F' and <i>equed</i> = 'R' or 'B', each element of <i>r</i> must be positive.</p> <p>If <i>equed</i> = 'C' or 'B', <i>A</i> is multiplied on the right by <i>diag</i>(<i>c</i>); if <i>equed</i> = 'N' or 'R', <i>c</i> is not accessed.</p> <p>If <i>fact</i> = 'F' and <i>equed</i> = 'C' or 'B', each element of <i>c</i> must be positive.</p> <p>Each element of <i>r</i> or <i>c</i> should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.</p>
<i>ldb</i>	The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ldx</i>	The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.
<i>n_err_bnds</i>	Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <i>err_bnds_norm</i> and <i>err_bnds_comp</i> descriptions in <i>Output Arguments</i> section below.
<i>nparams</i>	Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.

params

Array, size $\max(1, nparams)$. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params[0] : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
=1.0	Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

params[1] : Maximum number of residual computations allowed for refinement.

Default	10.0
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.

params[2] : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

Output Parameters

x

Array, size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout.

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the *original* system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the *equilibrated* system is:

$\text{inv}(\text{diag}(c)) * X$, if *trans* = 'N' and *equed* = 'C' or 'B'; or
 $\text{inv}(\text{diag}(r)) * X$, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.

ab

Array *ab* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *equed* ≠ 'N', *A* is scaled on exit as follows:

equed = 'R': $A = \text{diag}(r) * A$

equed = 'C': $A = A * \text{diag}(c)$

equed = 'B': $A = \text{diag}(r) * A * \text{diag}(c)$.

<i>afb</i>	If <i>fact</i> = 'N' or 'E', then <i>afb</i> is an output argument and on exit returns the factors <i>L</i> and <i>U</i> from the factorization $A = PLU$ of the original matrix <i>A</i> (if <i>fact</i> = 'N') or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E').
<i>b</i>	Overwritten by $diag(r)*B$ if <i>trans</i> = 'N' and <i>equed</i> = 'R' or 'B'; overwritten by <i>trans</i> = 'T' or 'C' and <i>equed</i> = 'C' or 'B'; not changed if <i>equed</i> = 'N'.
<i>r, c</i>	These arrays are output arguments if <i>fact</i> ≠ 'F'. Each element of these arrays is a power of the radix. See the description of <i>r, c</i> in <i>Input Arguments</i> section.
<i>rcond</i>	Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.
<i>rpvgrw</i>	Contains the reciprocal pivot growth factor: <div data-bbox="946 835 1057 882" data-label="Equation-Block"> $\ A\ /\ U\$ </div> <p>If this is much less than 1, the stability of the <i>LU</i> factorization of the (equilibrated) matrix <i>A</i> could be poor. This also means that the solution <i>X</i>, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \leq n$, this parameter contains the reciprocal pivot growth factor for the leading <i>info</i> columns of <i>A</i>. In ?gbsvx, this quantity is returned in <i>rpivot</i>.</p>
<i>berr</i>	Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes x_j an exact solution.
<i>err_bnds_norm</i>	Array of size <i>nrhs</i> * <i>n_err_bnds</i> . For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows: Normwise relative error in the <i>i</i> -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

<i>err</i> =1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
---------------	--

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3` Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error `err` is stored in `err_bnds_norm[(err-1)*nrhs + i - 1]`.

`err_bnds_comp`

Array of size `nrhs*n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params[2] = 0.0`), then `err_bnds_comp` is not accessed.

`err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and

$\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error `err` is stored in `err_bnds_comp[(err-1)*nrhs + i - 1]`.

`ipiv`

If `fact = 'N'` or `'E'`, then `ipiv` is an output argument and on exit contains the pivot indices from the factorization $A = P * L * U$ of the original matrix A (if `fact = 'N'`) or of the equilibrated matrix A (if `fact = 'E'`).

`equed`

If `fact != 'F'`, then `equed` is an output argument. It specifies the form of equilibration that was done (see the description of `equed` in *Input Arguments* section).

`params`

If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

`info`

If `info = 0`, the execution is successful. The solution to every right-hand side is guaranteed.

If `info = -i`, the i -th parameter had an illegal value.

If $0 < info \leq n$: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; `rcond = 0` is returned.

If `info = n+j`: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested `params[2] = 0.0`, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that `err_bnds_norm[j - 1] = 0.0` or `err_bnds_comp[j - 1] = 0.0`). See the definition of `err_bnds_norm` and `err_bnds_comp` for `err = 1`. To get information about all of the right-hand sides, check `err_bnds_norm` or `err_bnds_comp`.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, parameter *i* had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n*+*j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params*[2] = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that for column major layout *err_bnds_norm*[*j* - 1] = 0.0 or *err_bnds_comp*[*j* - 1] = 0.0; or for row major layout *err_bnds_norm*[(*j* - 1)**n_err_bnds*] = 0.0 or *err_bnds_comp*[(*j* - 1)**n_err_bnds*] = 0.0). See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?gtsv

Computes the solution to the system of linear equations with a tridiagonal coefficient matrix A and multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_sgtsv (int matrix_layout , lapack_int n , lapack_int nrhs , float *
dl , float * d , float * du , float * b , lapack_int ldb );

lapack_int LAPACKE_dgtsv (int matrix_layout , lapack_int n , lapack_int nrhs , double *
dl , double * d , double * du , double * b , lapack_int ldb );

lapack_int LAPACKE_cgtsv (int matrix_layout , lapack_int n , lapack_int nrhs ,
lapack_complex_float * dl , lapack_complex_float * d , lapack_complex_float * du ,
lapack_complex_float * b , lapack_int ldb );

lapack_int LAPACKE_zgtsv (int matrix_layout , lapack_int n , lapack_int nrhs ,
lapack_complex_double * dl , lapack_complex_double * d , lapack_complex_double * du ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for *X* the system of linear equations $A \cdot X = B$, where *A* is an *n*-by-*n* tridiagonal matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions. The routine uses Gaussian elimination with partial pivoting.

Note that the equation $A^T \cdot X = B$ may be solved by interchanging the order of the arguments *du* and *dl*.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>n</i>	The order of <i>A</i> , the number of rows in <i>B</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>dl</i>	The array <i>dl</i> (size $n - 1$) contains the $(n - 1)$ subdiagonal elements of <i>A</i> .
<i>d</i>	The array <i>d</i> (size n) contains the diagonal elements of <i>A</i> .
<i>du</i>	The array <i>du</i> (size $n - 1$) contains the $(n - 1)$ superdiagonal elements of <i>A</i> .
<i>b</i>	The array <i>b</i> of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>dl</i>	Overwritten by the $(n-2)$ elements of the second superdiagonal of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i> . These elements are stored in <i>dl</i> [0], ..., <i>dl</i> [$n - 3$].
<i>d</i>	Overwritten by the n diagonal elements of <i>U</i> .
<i>du</i>	Overwritten by the $(n-1)$ elements of the first superdiagonal of <i>U</i> .
<i>b</i>	Overwritten by the solution matrix <i>X</i> .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, $U_{i,i}$ is exactly zero, and the solution has not been computed. The factorization has not been completed unless $i = n$.

See Also

Matrix Storage Schemes

?gtsvx

*Computes the solution to the real or complex system of linear equations with a tridiagonal coefficient matrix *A* and multiple right-hand sides, and provides error bounds on the solution.*

Syntax

```
lapack_int LAPACKE_sgtsvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, const float* dl, const float* d, const float* du, float* dlf, float*
df, float* duf, float* du2, lapack_int* ipiv, const float* b, lapack_int ldb, float* x,
lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_dgtsvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, const double* dl, const double* d, const double* du, double* dlf,
double* df, double* duf, double* du2, lapack_int* ipiv, const double* b, lapack_int ldb,
double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

lapack_int LAPACKE_cgtsvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, const lapack_complex_float* dl, const lapack_complex_float* d, const
lapack_complex_float* du, lapack_complex_float* dlf, lapack_complex_float* df,
lapack_complex_float* duf, lapack_complex_float* du2, lapack_int* ipiv, const
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zgtsvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, const lapack_complex_double* dl, const lapack_complex_double* d, const
lapack_complex_double* du, lapack_complex_double* dlf, lapack_complex_double* df,
lapack_complex_double* duf, lapack_complex_double* du2, lapack_int* ipiv, const
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* rcond, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine uses the LU factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, $A^T \cdot X = B$, or $A^H \cdot X = B$, where A is a tridiagonal matrix of order n , the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gtsvx` performs the following steps:

1. If `fact = 'N'`, the LU decomposition is used to factor the matrix A as $A = L \cdot U$, where L is a product of permutation and unit lower bidiagonal matrices and U is an upper triangular matrix with nonzeros in only the main diagonal and first two superdiagonals.
2. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n + 1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>fact</code>	Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If $fact = 'F'$: on entry, dlf , df , duf , $du2$, and $ipiv$ contain the factored form of A ; arrays dl , d , du , dlf , df , duf , $du2$, and $ipiv$ will not be modified.

If $fact = 'N'$, the matrix A will be copied to dlf , df , and duf and factored.

trans

Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If $trans = 'N'$, the system has the form $A^*X = B$ (No transpose).

If $trans = 'T'$, the system has the form $A^T * X = B$ (Transpose).

If $trans = 'C'$, the system has the form $A^H * X = B$ (Conjugate transpose).

n

The number of linear equations, the order of the matrix A ; $n \geq 0$.

nrhs

The number of right hand sides, the number of columns of the matrices B and X ; $nrhs \geq 0$.

dl,d,du,dlf,df, duf,du2,b

Arrays:

dl, size $(n - 1)$, contains the subdiagonal elements of A .

d, size (n) , contains the diagonal elements of A .

du, size $(n - 1)$, contains the superdiagonal elements of A .

dlf, size $(n - 1)$. If $fact = 'F'$, then *dlf* is an input argument and on entry contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A as computed by [?gttrf](#).

df, size (n) . If $fact = 'F'$, then *df* is an input argument and on entry contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A .

duf, size $(n - 1)$. If $fact = 'F'$, then *duf* is an input argument and on entry contains the $(n - 1)$ elements of the first superdiagonal of U .

du2, size $(n - 2)$. If $fact = 'F'$, then *du2* is an input argument and on entry contains the $(n - 2)$ elements of the second superdiagonal of U .

b, size $\max(ldb * nrhs)$ for column major layout and $\max(ldb * n)$ for row major layout, contains the right-hand side matrix B .

ldb

The leading dimension of *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ldx

The leading dimension of *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

ipiv

Array, size at least $\max(1, n)$. If $fact = 'F'$, then *ipiv* is an input argument and on entry contains the pivot indices, as returned by [?gttrf](#).

Output Parameters

<i>x</i>	Array, size $\max(1, \text{ldx} * \text{nrhs})$ for column major layout and $\max(1, \text{ldx} * n)$ for row major layout. If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> .
<i>d1f</i>	If <i>fact</i> = 'N', then <i>d1f</i> is an output argument and on exit contains the (<i>n</i> -1) multipliers that define the matrix <i>L</i> from the <i>LU</i> factorization of <i>A</i> .
<i>df</i>	If <i>fact</i> = 'N', then <i>df</i> is an output argument and on exit contains the <i>n</i> diagonal elements of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i> .
<i>duf</i>	If <i>fact</i> = 'N', then <i>duf</i> is an output argument and on exit contains the (<i>n</i> -1) elements of the first superdiagonal of <i>U</i> .
<i>du2</i>	If <i>fact</i> = 'N', then <i>du2</i> is an output argument and on exit contains the (<i>n</i> -2) elements of the second superdiagonal of <i>U</i> .
<i>ipiv</i>	The array <i>ipiv</i> is an output argument if <i>fact</i> = 'N' and, on exit, contains the pivot indices from the factorization $A = L * U$; row <i>i</i> of the matrix was interchanged with row <i>ipiv</i> [<i>i</i> -1]. The value of <i>ipiv</i> [<i>i</i> -1] will always be <i>i</i> or <i>i</i> +1; <i>ipiv</i> [<i>i</i> -1]= <i>i</i> indicates a row interchange was not required.
<i>rcond</i>	An estimate of the reciprocal condition number of the matrix <i>A</i> . If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> >0.
<i>ferr</i>	Array, size at least $\max(1, \text{nrhs})$. Contains the estimated forward error bound for each solution vector x_j (the <i>j</i> -th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to x_j , <i>ferr</i> [<i>j</i> -1] is an estimated upper bound for the magnitude of the largest element in $x_j - x_{true}$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.
<i>berr</i>	Array, size at least $\max(1, \text{nrhs})$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes x_j an exact solution.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, and $i \leq n$, then $U_{i,i}$ is exactly zero. The factorization has not been completed unless $i = n$, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If *info* = *i*, and $i = n + 1$, then *U* is nonsingular, but *rcond* is less than machine precision,

meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

See Also

Matrix Storage Schemes

?dtsvb

Computes the solution to the system of linear equations with a diagonally dominant tridiagonal coefficient matrix A and multiple right-hand sides.

Syntax

```
void sdtsvb (const MKL_INT * n, const MKL_INT * nrhs, float * dl, float * d, const
float * du, float * b, const MKL_INT * ldb, MKL_INT * info );

void ddtsvb (const MKL_INT * n, const MKL_INT * nrhs, double * dl, double * d, const
double * du, double * b, const MKL_INT * ldb, MKL_INT * info );

void cdtsvb (const MKL_INT * n, const MKL_INT * nrhs, MKL_Complex8 * dl, MKL_Complex8 *
d, const MKL_Complex8 * du, MKL_Complex8 * b, const MKL_INT * ldb, MKL_INT * info );

void zdtsvb (const MKL_INT * n, const MKL_INT * nrhs, MKL_Complex16 * dl, MKL_Complex16 *
d, const MKL_Complex16 * du, MKL_Complex16 * b, const MKL_INT * ldb, MKL_INT *
info );
```

Include Files

- mkl.h

Description

The ?dtsvb routine solves a system of linear equations $A \cdot X = B$ for X , where A is an n -by- n diagonally dominant tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions. The routine uses the BABE (Burning At Both Ends) algorithm.

Note that the equation $A^T \cdot X = B$ may be solved by interchanging the order of the arguments *du* and *dl*.

Input Parameters

<i>n</i>	The order of A , the number of rows in B ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>dl</i> , <i>d</i> , <i>du</i> , <i>b</i>	<p>Arrays: <i>dl</i> (size $n - 1$), <i>d</i> (size n), <i>du</i> (size $n - 1$), <i>b</i>($\max(ldb \cdot nrhs)$ for column major layout and $\max(ldb \cdot n)$ for row major layout).</p> <p>The array <i>dl</i> contains the $(n - 1)$ subdiagonal elements of A.</p> <p>The array <i>d</i> contains the diagonal elements of A.</p> <p>The array <i>du</i> contains the $(n - 1)$ superdiagonal elements of A.</p> <p>The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations.</p>
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>dl</i>	Overwritten by the $(n-1)$ elements of the subdiagonal of the lower triangular matrices L_1, L_2 from the factorization of A (see dttfrfb).
<i>d</i>	Overwritten by the n diagonal element reciprocals of U .
<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	<p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p> <p>If <i>info</i> = <i>i</i>, u_{ii} is exactly zero, and the solution has not been computed. The factorization has not been completed unless $i = n$.</p>

Application Notes

A diagonally dominant tridiagonal system is defined such that $|d_i| > |dl_{i-1}| + |du_i|$ for any i :

$1 < i < n$, and $|d_1| > |du_1|$, $|d_n| > |dl_{n-1}|$

The underlying BABE algorithm is designed for diagonally dominant systems. Such systems have no numerical stability issue unlike the canonical systems that use elimination with partial pivoting (see [?gtsv](#)). The diagonally dominant systems are much faster than the canonical systems.

NOTE

- The current implementation of BABE has a potential accuracy issue on very small or large data close to the underflow or overflow threshold respectively. Scale the matrix before applying the solver in the case of such input data.
- Applying the [?dtsvb](#) factorization to non-diagonally dominant systems may lead to an accuracy loss, or false singularity detected due to no pivoting.

[?posv](#)

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix A and multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_sposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
float * a, lapack_int lda, float * b, lapack_int ldb);

lapack_int LAPACKE_dposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
double * a, lapack_int lda, double * b, lapack_int ldb);

lapack_int LAPACKE_cposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb);

lapack_int LAPACKE_dsposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
double * a, lapack_int lda, double * b, lapack_int ldb, double * x, lapack_int ldx,
lapack_int * iter);
```

```
lapack_int LAPACKE_zcposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb,
lapack_complex_double * x, lapack_int ldx, lapack_int * iter);
```

Include Files

- mkl.h

Description

The routine solves for X the real or complex system of linear equations $A * X = B$, where A is an n -by- n symmetric/Hermitian positive-definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if $uplo = 'U'$

or $A = L * L^T$ (real flavors) and $A = L * L^H$ (complex flavors), if $uplo = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $A * X = B$.

The `dsposv` and `zcposv` are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (`dsposv`) or single complex precision (`zcposv`) and use this factorization within an iterative refinement procedure to produce a solution with double precision (`dsposv`) / double complex precision (`zcposv`) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.

The iterative refinement is not going to be a winning strategy if the ratio single precision/complex performance over double precision/double complex performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to `ilaenv` in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if

`iter > itermax`

or for all the right-hand sides:

`rnrm < sqrt(n) * xnrm * anrm * eps * bwdmax`,

where

- `iter` is the number of the current iteration in the iterative refinement process
- `rnrm` is the infinity-norm of the residual
- `xnrm` is the infinity-norm of the solution
- `anrm` is the infinity-operator-norm of the matrix A
- `eps` is the machine epsilon returned by `dlamch` ('Epsilon').

The values `itermax` and `bwdmax` are fixed to 30 and 1.0d+00 respectively.

Input Parameters

`matrix_layout`

Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

`uplo`

Must be `'U'` or `'L'`.

Indicates whether the upper or lower triangular part of A is stored:

If `uplo = 'U'`, the upper triangle of A is stored.

	If <code>uplo = 'L'</code> , the lower triangle of A is stored.
<code>n</code>	The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<code>a, b</code>	Arrays: a (size $\max(1, lda)$), b , size $\max(ldb*nrhs)$ for column major layout and $\max(ldb*n)$ for row major layout,. The array a contains the upper or the lower triangular part of the matrix A (see <code>uplo</code>). Note that in the case of <code>zcpoenv</code> the imaginary parts of the diagonal elements need not be set and are assumed to be zero. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.
<code>lda</code>	The leading dimension of a ; $lda \geq \max(1, n)$.
<code>ldb</code>	The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<code>ldx</code>	The leading dimension of the array x ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

<code>a</code>	<p>If <code>info = 0</code>, the upper or lower triangular part of a is overwritten by the Cholesky factor U or L, as specified by <code>uplo</code>.</p> <p>If iterative refinement has been successfully used (<code>info= 0</code> and <code>iter</code> ≥ 0), then A is unchanged.</p> <p>If double precision factorization has been used (<code>info= 0</code> and <code>iter</code> < 0), then the array A contains the factors L or U from the Cholesky factorization.</p>
<code>b</code>	Overwritten by the solution matrix X .
<code>x</code>	Array, size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout. If <code>info = 0</code> , contains the n -by- $nrhs$ solution matrix X .
<code>iter</code>	<p>If <code>iter</code> < 0: iterative refinement has failed, double precision factorization has been performed</p> <ul style="list-style-type: none"> • If <code>iter</code> = -1: the routine fell back to full precision for implementation- or machine-specific reason • If <code>iter</code> = -2: narrowing the precision induced an overflow, the routine fell back to full precision • If <code>iter</code> = -3: failure of <code>spotrf</code> for <code>dsposv</code>, or <code>cpotrf</code> for <code>zcpoenv</code> • If <code>iter</code> = -31: stop the iterative refinement after the 30th iteration. <p>If <code>iter</code> > 0: iterative refinement has been successfully used. Returns the number of iterations.</p>

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

See Also

Matrix Storage Schemes

?posvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix A, and provides error bounds on the solution.

Syntax

```
lapack_int LAPACKES_posvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, char* equed,
float* s, float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr,
float* berr );
```

```
lapack_int LAPACKED_posvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, char* equed,
double* s, double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double*
ferr, double* berr );
```

```
lapack_int LAPACKEC_posvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, char* equed, float* s, lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
```

```
lapack_int LAPACKEZ_posvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, char* equed, double* s, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine uses the *Cholesky* factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where *A* is a *n*-by-*n* real symmetric/Hermitian positive definite matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?posvx performs the following steps:

1. If *fact* = 'E', real scaling factors *s* are computed to equilibrate the system:

$$\text{diag}(s)*A*\text{diag}(s)*\text{inv}(\text{diag}(s))*X = \text{diag}(s)*B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and B by $\text{diag}(s) * B$.

2. If $\text{fact} = 'N'$ or $'E'$, the Cholesky decomposition is used to factor the matrix A (after equilibration if $\text{fact} = 'E'$) as

$A = U^T * U$ (real), $A = U^H * U$ (complex), if $\text{uplo} = 'U'$,

or $A = L * L^T$ (real), $A = L * L^H$ (complex), if $\text{uplo} = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading i -by- i principal minor is not positive-definite, then the routine returns with $\text{info} = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $\text{info} = n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>fact</i>	<p>Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If $\text{fact} = 'F'$: on entry, af contains the factored form of A. If $\text{equed} = 'Y'$, the matrix A has been equilibrated with scaling factors given by s. a and af will not be modified.</p> <p>If $\text{fact} = 'N'$, the matrix A will be copied to af and factored.</p> <p>If $\text{fact} = 'E'$, the matrix A will be equilibrated if necessary, then copied to af and factored.</p>
<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored:</p> <p>If $\text{uplo} = 'U'$, the upper triangle of A is stored.</p> <p>If $\text{uplo} = 'L'$, the lower triangle of A is stored.</p>
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, the number of columns in B ; $\text{nrhs} \geq 0$.
<i>a</i> , <i>af</i> , <i>b</i>	Arrays: $a(\text{size max}(1, \text{lda} * n))$, $af(\text{size max}(1, \text{ldaf} * n))$, b , size $\text{max}(\text{ldb} * \text{nrhs})$ for column major layout and $\text{max}(\text{ldb} * n)$ for row major layout, .

The array *a* contains the matrix *A* as specified by *uplo*. If *fact* = 'F' and *equed* = 'Y', then *A* must have been equilibrated by the scaling factors in *s*, and *a* must contain the equilibrated matrix $diag(s) * A * diag(s)$.

The array *af* is an input argument if *fact* = 'F'. It contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix $diag(s) * A * diag(s)$.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

lda

The leading dimension of *a*; $lda \geq \max(1, n)$.

ldaf

The leading dimension of *af*; $ldaf \geq \max(1, n)$.

ldb

The leading dimension of *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

equed

Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

if *equed* = 'N', no equilibration was done (always true if *fact* = 'N');

if *equed* = 'Y', equilibration was done, that is, *A* has been replaced by $diag(s) * A * diag(s)$.

s

Array, size (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *equed* = 'N', *s* is not accessed.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

ldx

The leading dimension of the output array *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

x

Array, size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that if *equed* = 'Y', *A* and *B* are modified on exit, and the solution to the equilibrated system is $inv(diag(s)) * X$.

a

Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *fact* = 'E' and *equed* = 'Y', *A* is overwritten by $diag(s) * A * diag(s)$.

<i>af</i>	If <i>fact</i> = 'N' or 'E', then <i>af</i> is an output argument and on exit returns the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ (real routines), $A=U^H*U$ or $A=L*L^H$ (complex routines) of the original matrix <i>A</i> (if <i>fact</i> = 'N'), or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>a</i> for the form of the equilibrated matrix.
<i>b</i>	Overwritten by <i>diag(s)*B</i> , if <i>equed</i> = 'Y'; not changed if <i>equed</i> = 'N'.
<i>s</i>	This array is an output argument if <i>fact</i> ≠ 'F'. See the description of <i>s</i> in <i>Input Arguments</i> section.
<i>rcond</i>	An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector x_j (the <i>j</i> -th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to x_j , <i>ferr</i> [<i>j</i> -1] is an estimated upper bound for the magnitude of the largest element in $(x_j) - xtrue$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.
<i>berr</i>	Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes x_j an exact solution.
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, and $i \leq n$, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and $i = n + 1$, then *U* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

See Also

[Matrix Storage Schemes](#)

?posvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix A applying the Cholesky factorization.

Syntax

```
lapack_int LAPACKE_sposvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, char* equed,
float* s, float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float*
rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp,
lapack_int nparams, const float* params );
```

```
lapack_int LAPACKE_dposvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, char* equed,
double* s, double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double*
rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double*
err_bnds_comp, lapack_int nparams, const double* params );
```

```
lapack_int LAPACKE_cposvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, char* equed, float* s, lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* rcond, float* rpvgrw, float* berr,
lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams,
const float* params );
```

```
lapack_int LAPACKE_zposvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, char* equed, double* s, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr,
lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int
nparams, const double* params );
```

Include Files

- mkl.h

Description

The routine uses the *Cholesky* factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is an n -by- n real symmetric/Hermitian positive definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?posvxx performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

$$\text{diag}(s) * A * \text{diag}(s) * \text{inv}(\text{diag}(s)) * X = \text{diag}(s) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and B by $\text{diag}(s) * B$.

2. If $\text{fact} = 'N'$ or $'E'$, the Cholesky decomposition is used to factor the matrix A (after equilibration if $\text{fact} = 'E'$) as

$$A = U^T * U \text{ (real), } A = U^H * U \text{ (complex), if } \text{uplo} = 'U',$$

$$\text{or } A = L * L^T \text{ (real), } A = L * L^H \text{ (complex), if } \text{uplo} = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading i -by- i principal minor is not positive-definite, the routine returns with $\text{info} = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the rcond parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless $\text{params}[0]$ is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

fact

Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $\text{fact} = 'F'$, on entry, af contains the factored form of A . If equed is not 'N', the matrix A has been equilibrated with scaling factors given by s . Parameters a and af are not modified.

If $\text{fact} = 'N'$, the matrix A will be copied to af and factored.

If $\text{fact} = 'E'$, the matrix A will be equilibrated, if necessary, copied to af and factored.

uplo

Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored:

If $\text{uplo} = 'U'$, the upper triangle of A is stored.

If $\text{uplo} = 'L'$, the lower triangle of A is stored.

n

The number of linear equations; the order of the matrix A ; $n \geq 0$.

nrhs

The number of right-hand sides; the number of columns of the matrices B and X ; $\text{nrhs} \geq 0$.

a, af, b

Arrays: $a(\text{size max}(\text{lda}*n))$, $\text{af}(\text{size max}(\text{ldaf}*n))$, $b(\text{size max}(1, \text{ldb}*nrhs))$ for column major layout and $\text{max}(1, \text{ldb}*n)$ for row major layout).

The array *a* contains the matrix *A* as specified by *uplo*. If *fact* = 'F' and *equed* = 'Y', then *A* must have been equilibrated by the scaling factors in *s*, and *a* must contain the equilibrated matrix $diag(s) * A * diag(s)$.

The array *af* is an input argument if *fact* = 'F'. It contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *af* is the factored form of the equilibrated matrix $diag(s) * A * diag(s)$.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

lda

The leading dimension of the array *a*; $lda \geq \max(1, n)$.

ldaf

The leading dimension of the array *af*; $ldaf \geq \max(1, n)$.

equed

Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

if *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by $diag(s) * A * diag(s)$.

s

Array, size (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *equed* = 'N', *s* is not accessed.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

Each element of *s* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

The leading dimension of the array *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ldx

The leading dimension of the output array *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

n_err_bnds

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in the *Output Arguments* section below.

nparams

Specifies the number of parameters set in *params*. If ≤ 0 , the *params* array is never referenced and default values are used.

params

Array, size $\max(1, nparams)$. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed;

defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass `nparams = 0`, which prevents the source code from accessing the `params` argument.

`params[0]` : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

<code>=0.0</code>	No refinement is performed and no error bounds are computed.
<code>=1.0</code>	Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

`params[1]` : Maximum number of residual computations allowed for refinement.

Default	10.0
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy.

`params[2]` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

Output Parameters

<code>x</code>	Array, size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout. If <code>info = 0</code> , the array <code>x</code> contains the solution n -by- $nrhs$ matrix X to the original system of equations. Note that A and B are modified on exit if <code>equed ≠ 'N'</code> , and the solution to the equilibrated system is: $\text{inv}(\text{diag}(s)) * X.$
<code>a</code>	Array <code>a</code> is not modified on exit if <code>fact = 'F'</code> or <code>'N'</code> , or if <code>fact = 'E'</code> and <code>equed = 'N'</code> . If <code>fact = 'E'</code> and <code>equed = 'Y'</code> , A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.
<code>af</code>	If <code>fact = 'N'</code> or <code>'E'</code> , then <code>af</code> is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ (real routines), $A = U^H * U$ or $A = L * L^H$ (complex routines) of the original matrix A (if <code>fact = 'N'</code>), or of the equilibrated matrix A (if <code>fact = 'E'</code>). See the description of <code>a</code> for the form of the equilibrated matrix.
<code>b</code>	If <code>equed = 'N'</code> , B is not modified. If <code>equed = 'Y'</code> , B is overwritten by $\text{diag}(s) * B$.

s This array is an output argument if *fact*≠'F'. Each element of this array is a power of the radix. See the description of *s* in *Input Arguments* section.

rcond Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \leq n$, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

berr Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of *A* or *B* that makes x_j an exact solution.

err_bnds_norm Array of size *nrhs***n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

- | | |
|---------------|---|
| <i>err</i> =1 | "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. |
| <i>err</i> =2 | "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true. |
| <i>err</i> =3 | Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold |

$\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq \text{nrhs}$, and type of error err is stored in $\text{err_bnds_norm}[(\text{err}-1)*\text{nrhs} + i - 1]$.

`err_bnds_comp`

Array of size $\text{nrhs} * n_{\text{err_bnds}}$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{\text{true}_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}[2] = 0.0$), then `err_bnds_comp` is not accessed.

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is

"guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq \text{nrhs}$, and type of error err is stored in $\text{err_bnds_comp}[(\text{err}-1)*\text{nrhs} + i - 1]$.

equed

If $\text{fact} \neq 'F'$, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

params

If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

Return Values

This function returns a value *info*.

If $\text{info} = 0$, the execution is successful. The solution to every right-hand side is guaranteed.

If $\text{info} = -i$, parameter i had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $\text{rcond} = 0$ is returned.

If $\text{info} = n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $\text{params}[2] = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that for column major layout $\text{err_bnds_norm}[j - 1] = 0.0$ or $\text{err_bnds_comp}[j - 1] = 0.0$; or for row major layout $\text{err_bnds_norm}[(j - 1)*n_err_bnds] = 0.0$ or $\text{err_bnds_comp}[(j - 1)*n_err_bnds] = 0.0$). See the definition of *err_bnds_norm* and *err_bnds_comp* for $\text{err} = 1$. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?ppsv

Computes the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed coefficient matrix A and multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_sppsv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , float * ap , float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_dppsv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , double * ap , double * b , lapack_int ldb );
```



```
lapack_int LAPACKE_cppsv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , lapack_complex_float * ap , lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_zppsv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , lapack_complex_double * ap , lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n real symmetric/Hermitian positive-definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if $uplo = 'U'$

or $A = L * L^T$ (real flavors) and $A = L * L^H$ (complex flavors), if $uplo = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>ap, b</i>	Arrays: <i>ap</i> (size $\max(1, n*(n+1)/2)$), <i>b</i> , size $\max(ldb*nrhs)$ for column major layout and $\max(ldb*n)$ for row major layout,. The array <i>ap</i> contains the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in <i>packed storage</i> (see Matrix Storage Schemes). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>ap</i>	If <i>info</i> = 0, the upper or lower triangular part of A in packed storage is overwritten by the Cholesky factor U or L , as specified by <i>uplo</i> .
-----------	--

b Overwritten by the solution matrix X .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the leading minor of order *i* (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

See Also

Matrix Storage Schemes

?ppsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed coefficient matrix A , and provides error bounds on the solution.

Syntax

```
lapack_int LAPACKE_sppsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, float* ap, float* afp, char* equed, float* s, float* b, lapack_int ldb,
float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dppsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, double* ap, double* afp, char* equed, double* s, double* b, lapack_int
ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

```
lapack_int LAPACKE_cppsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float* ap, lapack_complex_float* afp, char* equed,
float* s, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int
ldx, float* rcond, float* ferr, float* berr );
```

```
lapack_int LAPACKE_zppsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_double* ap, lapack_complex_double* afp, char* equed,
double* s, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x,
lapack_int ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n -by- n symmetric or Hermitian positive-definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ppsvx performs the following steps:

1. If `fact = 'E'`, real scaling factors s are computed to equilibrate the system:

$$\text{diag}(s) * A * \text{diag}(s) * \text{inv}(\text{diag}(s)) * X = \text{diag}(s) * B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and B by $\text{diag}(s) * B$.

2. If `fact = 'N'` or `'E'`, the Cholesky decomposition is used to factor the matrix A (after equilibration if `fact = 'E'`) as

$$A = U^T * U \text{ (real)}, A = U^H * U \text{ (complex)}, \text{ if } \text{uplo} = 'U',$$

$$\text{or } A = L * L^T \text{ (real)}, A = L * L^H \text{ (complex)}, \text{ if } \text{uplo} = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading i -by- i principal minor is not positive-definite, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n+1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

`matrix_layout`

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

`fact`

Must be `'F'`, `'N'`, or `'E'`.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If `fact = 'F'`: on entry, `afp` contains the factored form of A . If `equed = 'Y'`, the matrix A has been equilibrated with scaling factors given by s .

`ap` and `afp` will not be modified.

If `fact = 'N'`, the matrix A will be copied to `afp` and factored.

If `fact = 'E'`, the matrix A will be equilibrated if necessary, then copied to `afp` and factored.

`uplo`

Must be `'U'` or `'L'`.

Indicates whether the upper or lower triangular part of A is stored:

If `uplo = 'U'`, the upper triangle of A is stored.

If `uplo = 'L'`, the lower triangle of A is stored.

`n`

The order of matrix A ; $n \geq 0$.

`nrhs`

The number of right-hand sides; the number of columns in B ; $nrhs \geq 0$.

`ap, afp, b`

Arrays: (size $\max(1, n*(n+1)/2)$, `afp` (size $\max(1, n*(n+1)/2)$, `b` of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout.

The array *ap* contains the upper or lower triangle of the original symmetric/Hermitian matrix *A* in *packed storage* (see [Matrix Storage Schemes](#)). In case when *fact* = 'F' and *equed* = 'Y', *ap* must contain the equilibrated matrix $\text{diag}(s) * A * \text{diag}(s)$.

The array *afp* is an input argument if *fact* = 'F' and contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *afp* is the factored form of the equilibrated matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldb The leading dimension of *b*; $\text{ldb} \geq \max(1, n)$ for column major layout and $\text{ldb} \geq \text{nrhs}$ for row major layout.

equed Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

if *equed* = 'N', no equilibration was done (always true if *fact* = 'N');

if *equed* = 'Y', equilibration was done, that is, *A* has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

s Array, size (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *equed* = 'N', *s* is not accessed.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

ldx The leading dimension of the output array *x*; $\text{ldx} \geq \max(1, n)$ for column major layout and $\text{ldx} \geq \text{nrhs}$ for row major layout.

Output Parameters

x Array, size $\max(1, \text{ldx} * \text{nrhs})$ for column major layout and $\max(1, \text{ldx} * n)$ for row major layout.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that if *equed* = 'Y', *A* and *B* are modified on exit, and the solution to the equilibrated system is $\text{inv}(\text{diag}(s)) * X$.

ap Array *ap* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *fact* = 'E' and *equed* = 'Y', *ap* is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

afp If *fact* = 'N' or 'E', then *afp* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ (real routines), $A = U^H * U$ or $A = L * L^H$ (complex

	routines) of the original matrix A (if $fact = 'N'$), or of the equilibrated matrix A (if $fact = 'E'$). See the description of ap for the form of the equilibrated matrix.
b	Overwritten by $diag(s)*B$, if $equed = 'Y'$; not changed if $equed = 'N'$.
s	This array is an output argument if $fact \neq 'F'$. See the description of s in <i>Input Arguments</i> section.
$rcond$	An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.
$ferr$	Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector x_j (the j -th column of the solution matrix X). If x_{true} is the true solution corresponding to x_j , $ferr[j-1]$ is an estimated upper bound for the magnitude of the largest element in $(x_j - x_{true})$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.
$berr$	Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of A or B that makes x_j an exact solution.
$equed$	If $fact \neq 'F'$, then $equed$ is an output argument. It specifies the form of equilibration that was done (see the description of $equed$ in <i>Input Arguments</i> section).

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, parameter i had an illegal value.

If $info = i$, and $i \leq n$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

See Also

[Matrix Storage Schemes](#)

?pbsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite band coefficient matrix A and multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_spbsv (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , lapack_int nrhs , float * ab , lapack_int ldab , float * b , lapack_int ldb );

lapack_int LAPACKE_dpbsv (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , lapack_int nrhs , double * ab , lapack_int ldab , double * b , lapack_int ldb );

lapack_int LAPACKE_cpbsv (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , lapack_int nrhs , lapack_complex_float * ab , lapack_int ldab ,
lapack_complex_float * b , lapack_int ldb );

lapack_int LAPACKE_zpbsv (int matrix_layout , char uplo , lapack_int n , lapack_int
kd , lapack_int nrhs , lapack_complex_double * ab , lapack_int ldab ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for X the real or complex system of linear equations $A * X = B$, where A is an n -by- n symmetric/Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if *uplo* = 'U'

or $A = L * L^T$ (real flavors) and $A = L * L^H$ (complex flavors), if *uplo* = 'L',

where U is an upper triangular band matrix and L is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as A . The factored form of A is then used to solve the system of equations $A * X = B$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>kd</i>	The number of superdiagonals of the matrix A if <i>uplo</i> = 'U', or the number of subdiagonals if <i>uplo</i> = 'L'; $kd \geq 0$.
<i>nrhs</i>	The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

<i>ab</i> , <i>b</i>	Arrays: <i>ab</i> (size max(1, <i>ldab</i> * <i>n</i>)), <i>b</i> of size max(1, <i>ldb</i> * <i>nrhs</i>) for column major layout and max(1, <i>ldb</i> * <i>n</i>) for row major layout. The array <i>ab</i> contains the upper or the lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i>) in <i>band storage</i> (see Matrix Storage Schemes). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
<i>ldab</i>	The leading dimension of the array <i>ab</i> ; <i>ldab</i> ≥ <i>kd</i> + 1.
<i>ldb</i>	The leading dimension of <i>b</i> ; <i>ldb</i> ≥ max(1, <i>n</i>) for column major layout and <i>ldb</i> ≥ <i>nrhs</i> for row major layout.

Output Parameters

<i>ab</i>	The upper or lower triangular part of <i>A</i> (in band storage) is overwritten by the Cholesky factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> , in the same storage format as <i>A</i> .
<i>b</i>	Overwritten by the solution matrix <i>X</i> .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the leading minor of order *i* (and therefore the matrix *A* itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

See Also

[Matrix Storage Schemes](#)

?pbsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive-definite band coefficient matrix A, and provides error bounds on the solution.

Syntax

```
lapack_int LAPACKE_spbsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int kd, lapack_int nrhs, float* ab, lapack_int ldab, float* afb, lapack_int
lda_fb, char* equed, float* s, float* b, lapack_int ldb, float* x, lapack_int ldx, float*
rcond, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dpbsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int kd, lapack_int nrhs, double* ab, lapack_int ldab, double* afb, lapack_int
lda_fb, char* equed, double* s, double* b, lapack_int ldb, double* x, lapack_int ldx,
double* rcond, double* ferr, double* berr );
```

```
lapack_int LAPACKE_cpbsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int kd, lapack_int nrhs, lapack_complex_float* ab, lapack_int ldab,
lapack_complex_float* afb, lapack_int lda_fb, char* equed, float* s,
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* rcond, float* ferr, float* berr );
```

```
lapack_int LAPACKE_zpbsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int kd, lapack_int nrhs, lapack_complex_double* ab, lapack_int ldab,
lapack_complex_double* afb, lapack_int ldafb, char* equed, double* s,
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* rcond, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine uses the Cholesky factorization $A=U^T*U$ (real flavors) / $A=U^H*U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where A is a n -by- n symmetric or Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?pbsvx` performs the following steps:

1. If `fact = 'E'`, real scaling factors s are computed to equilibrate the system:

$$\text{diag}(s)*A*\text{diag}(s)*\text{inv}(\text{diag}(s))*X = \text{diag}(s)*B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s)*A*\text{diag}(s)$ and B by $\text{diag}(s)*B$.

2. If `fact = 'N'` or `'E'`, the Cholesky decomposition is used to factor the matrix A (after equilibration if `fact = 'E'`) as

$$A = U^T*U \text{ (real)}, A = U^H*U \text{ (complex)}, \text{ if } \text{uplo} = 'U',$$

$$\text{or } A = L*L^T \text{ (real)}, A = L*L^H \text{ (complex)}, \text{ if } \text{uplo} = 'L',$$

where U is an upper triangular band matrix and L is a lower triangular band matrix.

3. If the leading i -by- i principal minor is not positive definite, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n+1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

`matrix_layout`

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

`fact`

Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If `fact = 'F'`: on entry, `afb` contains the factored form of A . If `equed = 'Y'`, the matrix A has been equilibrated with scaling factors given by s .

ab and *afb* will not be modified.

If *fact* = 'N', the matrix *A* will be copied to *afb* and factored.

If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *afb* and factored.

uplo

Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n

The order of matrix *A*; $n \geq 0$.

kd

The number of superdiagonals or subdiagonals in the matrix *A*; $kd \geq 0$.

nrhs

The number of right-hand sides, the number of columns in *B*; $nrhs \geq 0$.

ab, *afb*, *b*

Arrays: *ab*(size $\max(1, ldab*n)$), *afb*(size $\max(1, ldafb*n)$), *b* of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout.

The array *ab* contains the upper or lower triangle of the matrix *A* in *band storage* (see [Matrix Storage Schemes](#)).

If *fact* = 'F' and *equed* = 'Y', then *ab* must contain the equilibrated matrix $diag(s)*A*diag(s)$.

The array *afb* is an input argument if *fact* = 'F'. It contains the triangular factor *U* or *L* from the Cholesky factorization of the band matrix *A* in the same storage format as *A*. If *equed* = 'Y', then *afb* is the factored form of the equilibrated matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldab

The leading dimension of *ab*; $ldab \geq kd+1$.

ldaafb

The leading dimension of *afb*; $ldaafb \geq kd+1$.

ldb

The leading dimension of *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

equed

Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

if *equed* = 'N', no equilibration was done (always true if *fact* = 'N')

if *equed* = 'Y', equilibration was done, that is, *A* has been replaced by $diag(s)*A*diag(s)$.

s

Array, size (*n*). The array *s* contains the scale factors for *A*. This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *equed* = 'N', *s* is not accessed.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

ldx

The leading dimension of the output array *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

x

Array, size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the *original* system of equations. Note that if *equed* = 'Y', *A* and *B* are modified on exit, and the solution to the equilibrated system is $\text{inv}(\text{diag}(s)) * X$.

ab

On exit, if *fact* = 'E' and *equed* = 'Y', *A* is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

afb

If *fact* = 'N' or 'E', then *afb* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ (real routines), $A = U^H * U$ or $A = L * L^H$ (complex routines) of the original matrix *A* (if *fact* = 'N'), or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *ab* for the form of the equilibrated matrix.

b

Overwritten by $\text{diag}(s) * B$, if *equed* = 'Y'; not changed if *equed* = 'N'.

s

This array is an output argument if *fact* ≠ 'F'. See the description of *s* in *Input Arguments* section.

rcond

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector x_j (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to x_j , *ferr*[*j*-1] is an estimated upper bound for the magnitude of the largest element in $(x_j - x_{true})$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of *A* or *B* that makes x_j an exact solution.

equed

If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

Return Values

This function returns a value *info*.

If `info = 0`, the execution is successful.

If `info = -i`, parameter `i` had an illegal value.

If `info = i`, and $i \leq n$, the leading minor of order i (and therefore the matrix A itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; `rcond = 0` is returned. If `info = i`, and $i = n + 1$, then U is nonsingular, but `rcond` is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of `rcond` would suggest.

See Also

Matrix Storage Schemes

?ptsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite tridiagonal coefficient matrix A and multiple right-hand sides.

Syntax

```
lapack_int LAPACKESptsv( int matrix_layout, lapack_int n, lapack_int nrhs, float* d,
float* e, float* b, lapack_int ldb );
```

```
lapack_int LAPACKEdptsv( int matrix_layout, lapack_int n, lapack_int nrhs, double* d,
double* e, double* b, lapack_int ldb );
```

```
lapack_int LAPACKEcptsv( int matrix_layout, lapack_int n, lapack_int nrhs, float* d,
lapack_complex_float* e, lapack_complex_float* b, lapack_int ldb );
```

```
lapack_int LAPACKZptsv( int matrix_layout, lapack_int n, lapack_int nrhs, double* d,
lapack_complex_double* e, lapack_complex_double* b, lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for X the real or complex system of linear equations $A \cdot X = B$, where A is an n -by- n symmetric/Hermitian positive-definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

A is factored as $A = L \cdot D \cdot L^T$ (real flavors) or $A = L \cdot D \cdot L^H$ (complex flavors), and the factored form of A is then used to solve the system of equations $A \cdot X = B$.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>n</code>	The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

d	Array, dimension at least $\max(1, n)$. Contains the diagonal elements of the tridiagonal matrix A .
e, b	Arrays: e (size $n - 1$), b of size $\max(1, ldb * nrhs)$ for column major layout and $\max(1, ldb * n)$ for row major layout. The array e contains the $(n - 1)$ subdiagonal elements of A . The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.
ldb	The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

d	Overwritten by the n diagonal elements of the diagonal matrix D from the $L^*D^*L^T$ (real)/ $L^*D^*L^H$ (complex) factorization of A .
e	Overwritten by the $(n - 1)$ subdiagonal elements of the unit bidiagonal factor L from the factorization of A .
b	Overwritten by the solution matrix X .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, the leading minor of order *i* (and therefore the matrix A itself) is not positive-definite, and the solution has not been computed. The factorization has not been completed unless $i = n$.

See Also

Matrix Storage Schemes

?ptsvx

Uses factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite tridiagonal coefficient matrix A , and provides error bounds on the solution.

Syntax

```
lapack_int LAPACKE_sptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs,
const float* d, const float* e, float* df, float* ef, const float* b, lapack_int ldb,
float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs,
const double* d, const double* e, double* df, double* ef, const double* b, lapack_int
ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

```
lapack_int LAPACKE_cptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs,
const float* d, const lapack_complex_float* e, float* df, lapack_complex_float* ef,
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* rcond, float* ferr, float* berr );
```

```
lapack_int LAPACKE_zptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs,
const double* d, const lapack_complex_double* e, double* df, lapack_complex_double* ef,
const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int
ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine uses the Cholesky factorization $A = L^*D^*L^T$ (real)/ $A = L^*D^*L^H$ (complex) to compute the solution to a real or complex system of linear equations $A^*X = B$, where A is a n -by- n symmetric or Hermitian positive definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?ptsvx` performs the following steps:

1. If `fact = 'N'`, the matrix A is factored as $A = L^*D^*L^T$ (real flavors)/ $A = L^*D^*L^H$ (complex flavors), where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form $A = U^T*D*U$ (real flavors)/ $A = U^H*D*U$ (complex flavors).
2. If the leading i -by- i principal minor is not positive-definite, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n+1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>fact</code>	Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A is supplied on entry. If <code>fact = 'F'</code> : on entry, <code>df</code> and <code>ef</code> contain the factored form of A . Arrays <code>d</code> , <code>e</code> , <code>df</code> , and <code>ef</code> will not be modified. If <code>fact = 'N'</code> , the matrix A will be copied to <code>df</code> and <code>ef</code> , and factored.
<code>n</code>	The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<code>d, df</code>	Arrays: <code>d</code> (size n), <code>df</code> (size n). The array <code>d</code> contains the n diagonal elements of the tridiagonal matrix A . The array <code>df</code> is an input argument if <code>fact = 'F'</code> and on entry contains the n diagonal elements of the diagonal matrix D from the $L^*D^*L^T$ (real)/ $L^*D^*L^H$ (complex) factorization of A .

e, ef, b	<p>Arrays: e (size $n - 1$), ef (size $n - 1$), b, size $\max(ldb * nrhs)$ for column major layout and $\max(ldb * n)$ for row major layout. The array e contains the $(n - 1)$ subdiagonal elements of the tridiagonal matrix A.</p> <p>The array ef is an input argument if $fact = 'F'$ and on entry contains the $(n - 1)$ subdiagonal elements of the unit bidiagonal factor L from the $L * D * L^T$ (real)/ $L * D * L^H$ (complex) factorization of A.</p> <p>The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.</p>
ldb	The leading dimension of b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
ldx	The leading dimension of x ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

x	<p>Array, size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout.</p> <p>If $info = 0$ or $info = n + 1$, the array x contains the solution matrix X to the system of equations.</p>
df, ef	These arrays are output arguments if $fact = 'N'$. See the description of df, ef in <i>Input Arguments</i> section.
$rcond$	An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.
$ferr$	Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector x_j (the j -th column of the solution matrix X). If x_{true} is the true solution corresponding to x_j , $ferr_j$ is an estimated upper bound for the magnitude of the largest element in $(x_j - x_{true})$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.
$berr$	Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of A or B that makes x_j an exact solution.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter i had an illegal value.

If $info = i$, and $i \leq n$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

See Also

Matrix Storage Schemes

?ssysv

Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A and multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_ssysv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , float * a , lapack_int lda , lapack_int * ipiv , float * b , lapack_int ldb );

lapack_int LAPACKE_dsysv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , double * a , lapack_int lda , lapack_int * ipiv , double * b , lapack_int ldb );

lapack_int LAPACKE_csysv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , lapack_complex_float * a , lapack_int lda , lapack_int * ipiv ,
lapack_complex_float * b , lapack_int ldb );

lapack_int LAPACKE_zsysv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , lapack_complex_double * a , lapack_int lda , lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for X the real or complex system of linear equations $A * X = B$, where A is an n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U * D * U^T$ or $A = L * D * L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A * X = B$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	The order of matrix A ; $n \geq 0$.

<i>nrhs</i>	The number of right-hand sides; the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>a</i> , <i>b</i>	Arrays: <i>a</i> (size $\max(1, lda*n)$), <i>b</i> of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout. The array <i>a</i> contains the upper or the lower triangular part of the symmetric matrix <i>A</i> (see <i>uplo</i>). The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>a</i>	If <i>info</i> = 0, <i>a</i> is overwritten by the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>) from the factorization of <i>A</i> as computed by ?sytrf .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i> , as determined by ?sytrf . If <i>ipiv</i> [<i>i</i> -1] = <i>k</i> > 0, then <i>d_{ii}</i> is a 1-by-1 diagonal block, and the <i>i</i> -th row and column of <i>A</i> was interchanged with the <i>k</i> -th row and column. If <i>uplo</i> = 'U' and <i>ipiv</i> [<i>i</i>] = <i>ipiv</i> [<i>i</i> -1] = - <i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and (<i>i</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i> -th row and column. If <i>uplo</i> = 'L' and <i>ipiv</i> [<i>i</i>] = <i>ipiv</i> [<i>i</i> -1] = - <i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and (<i>i</i> +1)-th row and column of <i>A</i> was interchanged with the <i>m</i> -th row and column.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, *d_{ii}* is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

See Also

Matrix Storage Schemes

[?sysv_aa](#)

Computes the solution to a system of linear equations

$A * X = B$ for symmetric matrices.

```
lapack_int LAPACKE_ssysv_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, float * A, lapack_int lda, lapack_int * ipiv, float * B, lapack_int ldb);
```



```

lapack_int LAPACKE_dsysv_aa (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, double * A, lapack_int lda, lapack_int * ipiv, double * B, lapack_int ldb);

lapack_int LAPACKE_csysv_aa (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, lapack_complex_float * A, lapack_int lda, lapack_int * ipiv, lapack_complex_float
* B, lapack_int ldb);

lapack_int LAPACKE_zsysv_aa (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, lapack_complex_double * A, lapack_int lda, lapack_int * ipiv,
lapack_complex_double * B, lapack_int ldb);

```

Description

The `?sysv` routine computes the solution to a complex system of linear equations $A * X = B$, where A is an n -by- n symmetric matrix and X and B are n -by- $nrhs$ matrices.

Aasen's algorithm is used to factor A as $A = U * T * U^T$, if `uplo = 'U'`, or $A = L * T * L^T$, if `uplo = 'L'`, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and T is symmetric tri-diagonal. The factored form of A is then used to solve the system of equations $A * X = B$.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	<ul style="list-style-type: none"> <code>'U'</code>: The upper triangle of A is stored. <code>'L'</code>: The lower triangle of A is stored.
<code>n</code>	The number of linear equations; that is, the order of the matrix A . $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides; that is, the number of columns of the matrix B . $nrhs \geq 0$.
<code>A</code>	Array of size $\max(1, lda * n)$. On entry, the symmetric matrix A . If <code>uplo = 'U'</code> , the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced. If <code>uplo = 'L'</code> , the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.
<code>lda</code>	The leading dimension of the array A .
<code>B</code>	Array of size $\max(1, ldb * nrhs)$ for column-major layout and $\max(1, ldb * n)$ for row-major layout. On entry, the n -by- $nrhs$ right-hand side matrix B .
<code>ldb</code>	The leading dimension of the array B . $ldb \geq \max(1, n)$ for column-major layout and $ldb \geq nrhs$ for row-major layout.

Output Parameters

<code>A</code>	On exit, if <code>info = 0</code> , the tridiagonal matrix T and the multipliers used to obtain the factor U or L from the factorization $A = U * T * U^T$ or $A = L * T * L^T$ as computed by <code>?sytrf</code> .
<code>ipiv</code>	Array of size n . On exit, it contains the details of the interchanges; that is, the row and column k of A were interchanged with the row and column <code>ipiv(k)</code> .

B On exit, if $info = 0$, the n -by- $nrhs$ solution matrix X .

Return Values

This function returns a value $info$. $= 0$: Successful exit. < 0 : If $info = -i$, the i^{th} argument had an illegal value. > 0 : If $info = i$, $D(i,i)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution could not be computed.

?sysv_rook

Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A and multiple right-hand sides.

Syntax

```

lapack_int LAPACKE_ssysv_rook (int matrix_layout , char uplo , lapack_int n ,
lapack_int nrhs , float * a , lapack_int lda , lapack_int * ipiv , float * b ,
lapack_int ldb );

lapack_int LAPACKE_dsysv_rook (int matrix_layout , char uplo , lapack_int n ,
lapack_int nrhs , double * a , lapack_int lda , lapack_int * ipiv , double * b ,
lapack_int ldb );

lapack_int LAPACKE_csysv_rook (int matrix_layout , char uplo , lapack_int n ,
lapack_int nrhs , lapack_complex_float * a , lapack_int lda , lapack_int * ipiv ,
lapack_complex_float * b , lapack_int ldb );

lapack_int LAPACKE_zsysv_rook (int matrix_layout , char uplo , lapack_int n ,
lapack_int nrhs , lapack_complex_double * a , lapack_int lda , lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );

```

Include Files

- mkl.h

Description

The routine solves for X the real or complex system of linear equations $A * X = B$, where A is an n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U * D * U^T$ or $A = L * D * L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The `?sysv_rook` routine is called to compute the factorization of a complex symmetric matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method.

The factored form of A is then used to solve the system of equations $A * X = B$.

Input Parameters

 $matrix_layout$

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of <i>A</i> is stored:</p> <p>If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored.</p> <p>If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.</p>
<i>n</i>	The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>a</i> , <i>b</i>	<p>Arrays: <i>a</i>(size $\max(1, lda*n)$), <i>b</i> of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout.</p> <p>The array <i>a</i> contains the upper or the lower triangular part of the symmetric matrix <i>A</i> (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.</p>
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>a</i>	If <i>info</i> = 0, <i>a</i> is overwritten by the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>) from the factorization of <i>A</i> .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	<p>Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i>.</p> <p>If <i>ipiv</i>[<i>k</i> - 1] > 0, then rows and columns <i>k</i> and <i>ipiv</i>[<i>k</i> - 1] were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.</p> <p>If <i>uplo</i> = 'U' and <i>ipiv</i>[<i>k</i> - 1] < 0 and <i>ipiv</i>[<i>k</i> - 2] < 0, then rows and columns <i>k</i> and -<i>ipiv</i>[<i>k</i> - 1] were interchanged, rows and columns <i>k</i> - 1 and -<i>ipiv</i>[<i>k</i> - 2] were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.</p> <p>If <i>uplo</i> = 'L' and <i>ipiv</i>[<i>k</i> - 1] < 0 and <i>ipiv</i>[<i>k</i>] < 0, then rows and columns <i>k</i> and -<i>ipiv</i>[<i>k</i> - 1] were interchanged, rows and columns <i>k</i> + 1 and -<i>ipiv</i>[<i>k</i>] were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.</p>

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

See Also

Matrix Storage Schemes

?sysv_rk

*Computes the solution to system of linear equations $A * X = B$ for SY matrices.*

```
lapack_int LAPACKE_ssysv_rk (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, float * A, lapack_int lda, float * e, lapack_int * ipiv, float * B, lapack_int
ldb);

lapack_int LAPACKE_dsysv_rk (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, double * A, lapack_int lda, double * e, lapack_int * ipiv, double * B, lapack_int
ldb);

lapack_int LAPACKE_csysv_rk (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, lapack_complex_float * A, lapack_int lda, lapack_complex_float * e, lapack_int *
ipiv, lapack_complex_float * B, lapack_int ldb);

lapack_int LAPACKE_zsysv_rk (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, lapack_complex_double * A, lapack_int lda, lapack_complex_double * e, lapack_int
* ipiv, lapack_complex_double * B, lapack_int ldb);
```

Description

?sysv_rk computes the solution to a real or complex system of linear equations $A * X = B$, where A is an n -by- n symmetric matrix and X and B are n -by- $nrhs$ matrices.

The bounded Bunch-Kaufman (rook) diagonal pivoting method is used to factor A as $A = P * U * D * (U^T) * (P^T)$, if $uplo = 'U'$, or $A = P * L * D * (L^T) * (P^T)$, if $uplo = 'L'$, where U (or L) is unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of U (or L), P is a permutation matrix, P^T is the transpose of P , and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

?sytrf_rk is called to compute the factorization of a real or complex symmetric matrix. The factored form of A is then used to solve the system of equations $A * X = B$ by calling BLAS3 routine ?sytrs_3.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the symmetric matrix A is stored: <ul style="list-style-type: none"> = 'U': The upper triangle of A is stored. = 'L': The lower triangle of A is stored.
<i>n</i>	The number of linear equations; that is, the order of the matrix A . $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; that is, the number of columns of the matrix B . $nrhs \geq 0$.
<i>A</i>	Array of size $\max(1, lda * n)$. On entry, the symmetric matrix A . If $uplo = 'U'$, the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is

not referenced. If `uplo = 'L'`, the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.

`lda` The leading dimension of the array A .

`B` Array of size $\max(1, ldb * nrhs)$. On entry, the n -by- $nrhs$ right-hand side matrix B .

`ldb` The leading dimension of the array B . $ldb \geq \max(1, n)$ for column-major layout and $ldb \geq nrhs$ for row-major layout.

Output Parameters

`A` On exit, if `info = 0`, the diagonal of the block diagonal matrix D and factors U or L as computed by `?sytrf_rk`:

- Only diagonal elements of the symmetric block diagonal matrix D on the diagonal of A ; that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of D are stored on exit in array `e`.
- If `uplo = 'U'`, factor U in the superdiagonal part of A . If `uplo = 'L'`, factor L in the subdiagonal part of A . For more information, see the description of the `?sytrf_rk` routine.

`e` Array of size n . On exit, contains the output computed by the factorization routine `?sytrf_rk`; that is, the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If `uplo = 'U'`, $e(i) = D(i-1,i)$, $i=1:N-1$, and $e(1)$ is set to 0. If `uplo = 'L'`, $e(i) = D(i+1,i)$, $i=1:N-1$, and $e(n)$ is set to 0.

NOTE For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e(k)$ is set to 0 in both the `uplo = 'U'` and `uplo = 'L'` cases. For more information, see the description of the `?sytrf_rk` routine.

`ipiv` Array of size n . Details of the interchanges and the block structure of D , as determined by `?sytrf_rk`. For more information, see the description of the `?sytrf_rk` routine.

`B` On exit, if `info = 0`, the n -by- $nrhs$ solution matrix X .

Return Values

This function returns a value `info`.

= 0: Successful exit.

< 0: If `info = -k`, the k^{th} argument had an illegal value.

> 0: If `info = k`, the matrix A is singular. If `uplo = 'U'`, column k in the upper triangular part of A contains all zeros. If `uplo = 'L'`, column k in the lower triangular part of A contains all zeros. Therefore $D(k,k)$ is exactly zero, and superdiagonal elements of column k of U (or subdiagonal elements of column k of L) are all zeros. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?sysvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric coefficient matrix A , and provides error bounds on the solution.

Syntax

```
lapack_int LAPACKE_ssysvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const float* a, lapack_int lda, float* af, lapack_int ldaf,
lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float*
rcond, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dsysvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const double* a, lapack_int lda, double* af, lapack_int ldaf,
lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double*
rcond, double* ferr, double* berr );
```

```
lapack_int LAPACKE_csysvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, lapack_complex_float*
af, lapack_int ldaf, lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
```

```
lapack_int LAPACKE_zsysvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, lapack_complex_double*
af, lapack_int ldaf, lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is a n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?sysvx performs the following steps:

1. If $fact = 'N'$, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U \cdot D \cdot U^T$ or $A = L \cdot D \cdot L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

<i>fact</i>	<p>Must be 'F' or 'N'.</p> <p>Specifies whether or not the factored form of the matrix <i>A</i> has been supplied on entry.</p> <p>If <i>fact</i> = 'F': on entry, <i>af</i> and <i>ipiv</i> contain the factored form of <i>A</i>. Arrays <i>a</i>, <i>af</i>, and <i>ipiv</i> will not be modified.</p> <p>If <i>fact</i> = 'N', the matrix <i>A</i> will be copied to <i>af</i> and factored.</p>
<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of <i>A</i> is stored:</p> <p>If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored.</p> <p>If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.</p>
<i>n</i>	The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>a</i> , <i>af</i> , <i>b</i>	<p>Arrays: <i>a</i>(size $\max(1, lda*n)$), <i>af</i>(size $\max(1, ldaf*n)$), <i>b</i> of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout .</p> <p>The array <i>a</i> contains the upper or the lower triangular part of the symmetric matrix <i>A</i> (see <i>uplo</i>).</p> <p>The array <i>af</i> is an input argument if <i>fact</i> = 'F'. It contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by ?sytrf.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.</p>
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ipiv</i>	<p>Array, size at least $\max(1, n)$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. It contains details of the interchanges and the block structure of <i>D</i>, as determined by ?sytrf.</p> <p>If $ipiv[i-1] = k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the <i>i</i>-th row and column of <i>A</i> was interchanged with the <i>k</i>-th row and column.</p> <p>If <i>uplo</i> = 'U' and $ipiv[i] = ipiv[i-1] = -m < 0$, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i+1</i>, and (<i>i</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i>-th row and column.</p> <p>If <i>uplo</i> = 'L' and $ipiv[i] = ipiv[i-1] = -m < 0$, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i+1</i>, and (<i>i+1</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i>-th row and column.</p>

ldx The leading dimension of the output array *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

x Array, size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout.
If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations.

af, ipiv These arrays are output arguments if *fact* = 'N'.
See the description of *af, ipiv* in *Input Arguments* section.

rcond An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector x_j (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to x_j , *ferr*[*j*-1] is an estimated upper bound for the magnitude of the largest element in $(x_j - x_{true})$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of *A* or *B* that makes x_j an exact solution.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and $i = n + 1$, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

See Also

[Matrix Storage Schemes](#)

?sysvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a symmetric indefinite coefficient matrix A applying the diagonal pivoting factorization.

Syntax

```
lapack_int LAPACKE_ssysvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, lapack_int* ipiv,
char* equed, float* s, float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond,
float* rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float*
err_bnds_comp, lapack_int nparams, const float* params );
```

```
lapack_int LAPACKE_dsysvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, lapack_int*
ipiv, char* equed, double* s, double* b, lapack_int ldb, double* x, lapack_int ldx,
double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds, double*
err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );
```

```
lapack_int LAPACKE_csysvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, float* s, lapack_complex_float* b,
lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* rpvgrw,
float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp,
lapack_int nparams, const float* params );
```

```
lapack_int LAPACKE_zsysvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, double* s, lapack_complex_double* b,
lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double*
rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double*
err_bnds_comp, lapack_int nparams, const double* params );
```

Include Files

- mkl.h

Description

The routine uses the *diagonal pivoting* factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is an n -by- n real symmetric/Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine `?sysvxx` performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

$$\text{diag}(s) * A * \text{diag}(s) * \text{inv}(\text{diag}(s)) * X = \text{diag}(s) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and B by $\text{diag}(s) * B$.

2. If *fact* = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as

$A = U^*D^*U^T$, if $uplo = 'U'$,
 or $A = L^*D^*L^T$, if $uplo = 'L'$,

where U or L is a product of permutation and unit upper (lower) triangular matrices, and D is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

3. If some $D(i, i) = 0$, so that D is exactly singular, the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the $rcond$ parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless $params[0]$ is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $diag(r)$ so that it solves the original system before equilibration.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>fact</i>	<p>Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If $fact = 'F'$, on entry, af and $ipiv$ contain the factored form of A. If $equed$ is not 'N', the matrix A has been equilibrated with scaling factors given by s. Parameters a, af, and $ipiv$ are not modified.</p> <p>If $fact = 'N'$, the matrix A will be copied to af and factored.</p> <p>If $fact = 'E'$, the matrix A will be equilibrated, if necessary, copied to af and factored.</p>
<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored:</p> <p>If $uplo = 'U'$, the upper triangle of A is stored.</p> <p>If $uplo = 'L'$, the lower triangle of A is stored.</p>
<i>n</i>	The number of linear equations; the order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<i>a</i> , <i>af</i> , <i>b</i>	<p>Arrays: a(size $\max(1, lda*n)$), af(size $\max(1, ldaf*n)$), b, size $\max(ldb*nrhs)$ for column major layout and $\max(lb*n)$ for row major layout,.</p> <p>The array a contains the symmetric matrix A as specified by $uplo$. If $uplo = 'U'$, the leading n-by-n upper triangular part of a contains the upper triangular part of the matrix A and the strictly lower triangular part of a is not referenced. If $uplo = 'L'$, the leading n-by-n lower triangular part of a contains the lower triangular part of the matrix A and the strictly upper triangular part of a is not referenced.</p>

The array *af* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* and *L* from the factorization $A = U^*D^*U^T$ or $A = L^*D^*L^T$ as computed by `?sytrf`.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

lda

The leading dimension of the array *a*; $lda \geq \max(1, n)$.

ldaf

The leading dimension of the array *af*; $ldaf \geq \max(1, n)$.

ipiv

Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D* as determined by `?sytrf`. If *ipiv*[*k*-1] > 0, rows and columns *k* and *ipiv*[*k*-1] were interchanged and *D*(*k*,*k*) is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*[*i*] = *ipiv*[*i* - 1] = *m* < 0, *D* has a 2-by-2 diagonal block in rows and columns *i* and *i* + 1, and the *i*-th row and column of *A* were interchanged with the *m*-th row and column.

If *uplo* = 'L' and *ipiv*[*i*] = *ipiv*[*i* - 1] = *m* < 0, *D* has a 2-by-2 diagonal block in rows and columns *i* and *i* + 1, and the (*i* + 1)-st row and column of *A* were interchanged with the *m*-th row and column.

equed

Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

if *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by $diag(s) * A * diag(s)$.

s

Array, size (*n*). The array *s* contains the scale factors for *A*. If *equed* = 'Y', *A* is multiplied on the left and right by $diag(s)$.

This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

Each element of *s* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

The leading dimension of the array *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ldx

The leading dimension of the output array *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

<i>n_err_bnds</i>	Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <i>err_bnds_norm</i> and <i>err_bnds_comp</i> descriptions in the <i>Output Arguments</i> section below.								
<i>nparams</i>	Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.								
<i>params</i>	<p>Array, size $\max(1, nparams)$. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument.</p> <p><i>params</i>[0] : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).</p> <table> <tr> <td>=0.0</td><td>No refinement is performed and no error bounds are computed.</td></tr> <tr> <td>=1.0</td><td>Use the extra-precise refinement algorithm.</td></tr> </table> <p>(Other values are reserved for future use.)</p> <p><i>params</i>[1] : Maximum number of residual computations allowed for refinement.</p> <table> <tr> <td>Default</td><td>10.0</td></tr> <tr> <td>Aggressive</td><td>Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i>. If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.</td></tr> </table> <p><i>params</i>[2] : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).</p>	=0.0	No refinement is performed and no error bounds are computed.	=1.0	Use the extra-precise refinement algorithm.	Default	10.0	Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.
=0.0	No refinement is performed and no error bounds are computed.								
=1.0	Use the extra-precise refinement algorithm.								
Default	10.0								
Aggressive	Set to 100.0 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.								

Output Parameters

<i>x</i>	<p>Array, size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout).</p> <p>If <i>info</i> = 0, the array <i>x</i> contains the solution <i>n</i>-by-<i>nrhs</i> matrix <i>X</i> to the original system of equations. Note that <i>A</i> and <i>B</i> are modified on exit if <i>equed</i> ≠ 'N', and the solution to the equilibrated system is:</p> $\text{inv}(\text{diag}(s)) * X.$
<i>a</i>	If <i>fact</i> = 'E' and <i>equed</i> = 'Y', overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

<i>af</i>	If <i>fact</i> = 'N', <i>af</i> is an output argument and on exit returns the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> from the factorization $A = U^*D^*U^T$ or $A = L^*D^*L^T$.
<i>b</i>	If <i>equed</i> = 'N', <i>B</i> is not modified. If <i>equed</i> = 'Y', <i>B</i> is overwritten by $diag(s) * B$.
<i>s</i>	This array is an output argument if <i>fact</i> ≠ 'F'. Each element of this array is a power of the radix. See the description of <i>s</i> in <i>Input Arguments</i> section.
<i>rcond</i>	Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \leq n$, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

berr Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of *A* or *B* that makes x_j an exact solution.

err_bnds_norm Array of size *nrhs***n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. Up to three pieces of information are returned.

<i>err</i> =1	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
<i>err</i> =2	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and

$\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error `err` is stored in `err_bnds_norm[(err-1)*nrhs + i - 1]`.

`err_bnds_comp`

Array of size `nrhs*n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params[2] = 0.0`), then `err_bnds_comp` is not accessed.

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq \text{nrhs}$, and type of error *err* is stored in `err_bnds_comp[(err-1)*nrhs + i - 1]`.

ipiv

If *fact* = 'N', *ipiv* is an output argument and on exit contains details of the interchanges and the block structure D , as determined by `ssytrf` for single precision flavors and `dsytrf` for double precision flavors.

equed

If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

params

If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, parameter *i* had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = $n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested `params[2] = 0.0`, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that for column major layout `err_bnds_norm[j - 1] = 0.0` or `err_bnds_comp[j - 1] = 0.0`; or for row major layout `err_bnds_norm[(j - 1)*n_err_bnds] = 0.0` or `err_bnds_comp[(j - 1)*n_err_bnds] = 0.0`). See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

[Matrix Storage Schemes](#)

?hesv

Computes the solution to the system of linear equations with a Hermitian matrix A and multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_chesv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , lapack_complex_float * a , lapack_int lda , lapack_int * ipiv ,
lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_zhesv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , lapack_complex_double * a , lapack_int lda , lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for X the complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U^*D^*U^H$ or $A = L^*D^*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A , and A is factored as $U^*D^*U^H$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A , and A is factored as $L^*D^*L^H$.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>a</i> , <i>b</i>	Arrays: <i>a</i> (size $\max(1, lda*n)$), <i>b</i> of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout. The array <i>a</i> contains the upper or the lower triangular part of the Hermitian matrix A (see <i>uplo</i>). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations.

<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>a</i>	If <i>info</i> = 0, <i>a</i> is overwritten by the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>) from the factorization of <i>A</i> as computed by ?hetrf .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	<p>Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i>, as determined by ?hetrf.</p> <p>If <i>ipiv</i>[<i>i</i>-1] = <i>k</i> > 0, then <i>d_{ii}</i> is a 1-by-1 diagonal block, and the <i>i</i>-th row and column of <i>A</i> was interchanged with the <i>k</i>-th row and column.</p> <p>If <i>uplo</i> = 'U' and <i>ipiv</i>[<i>i</i>] = <i>ipiv</i>[<i>i</i>-1] = -<i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i>+1, and (<i>i</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i>-th row and column.</p> <p>If <i>uplo</i> = 'L' and <i>ipiv</i>[<i>i</i>] = <i>ipiv</i>[<i>i</i>-1] = -<i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i>+1, and (<i>i</i>+1)-th row and column of <i>A</i> was interchanged with the <i>m</i>-th row and column.</p>

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, *d_{ii}* is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

See Also

Matrix Storage Schemes

[?hesv_aa](#)

Computes the solution to system of linear equations for HE matrices.

```
LAPACK_DECL lapack_int LAPACKE_chesv_aa (int matrix_layout, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float * a, lapack_int lda, lapack_int * ipiv,
lapack_complex_float * b, lapack_int ldb );
```

```
LAPACK_DECL lapack_int LAPACKE_chesv_aa_work (int matrix_layout, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float * a, lapack_int lda, lapack_int * ipiv,
lapack_complex_float * b, lapack_int ldb, lapack_complex_float * work, lapack_int lwork );
```

Description

[?hesv_aa](#) computes the solution to a complex system of linear equations $A * X = B$, where *A* is an *n*-by-*n* Hermitian matrix and *X* and *B* are *n*-by-*nrhs* matrices. Aasen's algorithm is used to factor *A* as

$A = U * T * U^H$ if *uplo* = 'U', or

$A = L * T * L^H$ if $uplo = 'L'$,

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and T is Hermitian and tridiagonal. The factored form of A is then used to solve the system of equations $A * X = B$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	If <i>uplo</i> = 'U': The upper triangle of A is stored. If <i>uplo</i> = 'L': the lower triangle of A is stored.
<i>n</i>	The number of linear equations or the order of the matrix A . $n \geq 0$.
<i>nrhs</i>	The number of right hand sides or the number of columns of the matrix B . $nrhs \geq 0$.
<i>a</i>	Array of size $lda*n$. On entry, the Hermitian matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.
<i>lda</i>	The leading dimension of the array a . $lda \geq \max(1, n)$.
<i>b</i>	Array of size $ldb*nrhs$. On entry, the n -by- $nrhs$ right hand side matrix B .
<i>ldb</i>	The leading dimension of the array b . $ldb \geq \max(1, n)$.
<i>lwork</i>	The length of <i>work</i> . $lwork \geq \max(1, 2*n, 3*n-2)$, and for best performance $lwork \geq \max(1, n*nb)$, where nb is the optimal blocksize for ?hetrf. If $lwork < n$, TRS is done with Level BLAS 2. If $lwork \geq n$, TRS is done with Level BLAS 3. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla.

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, the tridiagonal matrix T and the multipliers used to obtain the factor U or L from the factorization $A = U*T*U^H$ or $A = L*T*L^H$ as computed by ?hetrf_aa.
<i>ipiv</i>	Array of size (n) On exit, it contains the details of the interchanges: row and column k of A were interchanged with the row and column $ipiv[k]$.
<i>b</i>	On exit, if <i>info</i> = 0, the n -by- $nrhs$ solution matrix X .
<i>work</i>	Array of size ($\max(1, lwork)$). On exit, if <i>info</i> = 0, <i>work</i> [0] returns the optimal <i>lwork</i> .

Return Values

This function returns a value *info*.

If *info* = 0: successful exit.

If *info* < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

If *info* > 0: if *info* = *i*, $D_{i,i}$ is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution could not be computed.

?hesv_rk

*?hesv_rk computes the solution to a system of linear equations $A * X = B$ for Hermitian matrices.*

```
lapack_int LAPACKE_chesv_rk (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, lapack_complex_float * A, lapack_int lda, lapack_complex_float * e, lapack_int *
ipiv, lapack_complex_float * B, lapack_int ldb);
```

```
lapack_int LAPACKE_zhesv_rk (int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, lapack_complex_double * A, lapack_int lda, lapack_complex_double * e, lapack_int
* ipiv, lapack_complex_double * B, lapack_int ldb);
```

Description

?hesv_rk computes the solution to a complex system of linear equations $A * X = B$, where *A* is an *n*-by-*n* Hermitian matrix and *X* and *B* are *n*-by-*nrhs* matrices.

The bounded Bunch-Kaufman (rook) diagonal pivoting method is used to factor *A* as $A = P * U * D * (U^H) * (P^T)$, if *uplo* = 'U', or $A = P * L * D * (L^H) * (P^T)$, if *uplo* = 'L', where *U* (or *L*) is unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of *U* (or *L*), *P* is a permutation matrix, P^T is the transpose of *P*, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

?hetrf_rk is called to compute the factorization of a complex Hermitian matrix. The factored form of *A* is then used to solve the system of equations $A * X = B$ by calling BLAS3 routine *?hetrs_3*.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is stored: <ul style="list-style-type: none"> = 'U': The upper triangle of <i>A</i> is stored. = 'L': The lower triangle of <i>A</i> is stored.
<i>n</i>	The number of linear equations; that is, the order of the matrix <i>A</i> . $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; that is, the number of columns of the matrix <i>B</i> . $nrhs \geq 0$.
<i>A</i>	Array of size $\max(1, lda * n)$. On entry, the Hermitian matrix <i>A</i> . If <i>uplo</i> = 'U': the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>A</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>A</i> is not referenced. If <i>uplo</i> = 'L': the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>A</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>A</i> is not referenced.
<i>lda</i>	The leading dimension of the array <i>A</i> .

<i>B</i>	On entry, the <i>n</i> -by- <i>nrhs</i> right-hand side matrix <i>B</i> . The size of <i>B</i> is $\max(1, \text{ldb} * \text{nrhs})$ for column-major layout and $\max(1, \text{ldb} * n)$ for row-major layout.
<i>ldb</i>	The leading dimension of the array <i>B</i> . $\text{ldb} \geq \max(1, n)$ for column-major layout and $\text{ldb} \geq \text{nrhs}$ for row-major layout.

Output Parameters

<i>A</i>	On exit, if <i>info</i> = 0, diagonal of the block diagonal matrix <i>D</i> and factors <i>U</i> or <i>L</i> as computed by ?hetrf_rk: <ul style="list-style-type: none"> Only diagonal elements of the Hermitian block diagonal matrix <i>D</i> on the diagonal of <i>A</i>; that is, $D(k,k) = A(k,k)$; (superdiagonal (or subdiagonal) elements of <i>D</i> are stored on exit in array <i>e</i>). <p>—and—</p> <ul style="list-style-type: none"> If <i>uplo</i> = 'U', factor <i>U</i> in the superdiagonal part of <i>A</i>. If <i>uplo</i> = 'L', factor <i>L</i> in the subdiagonal part of <i>A</i>. <p>For more information, see the description of the ?hetrf_rk routine.</p>
<i>e</i>	Array of size <i>n</i> . On exit, contains the output computed by the factorization routine ?hetrf_rk; that is, the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix <i>D</i> with 1-by-1 or 2-by-2 diagonal blocks: <ul style="list-style-type: none"> If <i>uplo</i> = 'U', $e(i) = D(i-1,i)$, $i=2:N$, $e(1)$ is set to 0. If <i>uplo</i> = 'L', $e(i) = D(i+1,i)$, $i=1:N-1$, $e(n)$ is set to 0.

NOTE For a 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e(k)$ is set to 0 in both the *uplo* = 'U' and *uplo* = 'L' cases.

	For more information, see the description of the ?hetrf_rk routine.
<i>ipiv</i>	Array of size <i>n</i> . Details of the interchanges and the block structure of <i>D</i> , as determined by ?hetrf_rk.
<i>B</i>	On exit, if <i>info</i> = 0, the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> .

Return Values

This function returns a value *info*.

= 0: Successful exit.

< 0: If *info* = -*k*, the *k*th argument had an illegal value.

> 0: If *info* = *k*, the matrix *A* is singular. If *uplo* = 'U', column *k* in the upper triangular part of *A* contains all zeros. If *uplo* = 'L', column *k* in the lower triangular part of *A* contains all zeros. Therefore $D(k,k)$ is exactly zero, and superdiagonal elements of column *k* of *U* (or subdiagonal elements of column *k* of *L*) are all zeros. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?hesvx

Uses the diagonal pivoting factorization to compute the solution to the complex system of linear equations with a Hermitian coefficient matrix A , and provides error bounds on the solution.

Syntax

```
lapack_int LAPACKE_chesvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, lapack_complex_float*
af, lapack_int ldaf, lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zhesvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, lapack_complex_double*
af, lapack_int ldaf, lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A \cdot X = B$, where A is an n -by- n Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?hesvx performs the following steps:

1. If $fact = 'N'$, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U \cdot D \cdot U^H$ or $A = L \cdot D \cdot L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>fact</i>	Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If $fact = 'F'$: on entry, af and $ipiv$ contain the factored form of A . Arrays a , af , and $ipiv$ are not modified. If $fact = 'N'$, the matrix A is copied to af and factored.

<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of <i>A</i> is stored and how <i>A</i> is factored:</p> <p>If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the Hermitian matrix <i>A</i>, and <i>A</i> is factored as $U^*D^*U^H$.</p> <p>If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the Hermitian matrix <i>A</i>; <i>A</i> is factored as $L^*D^*L^H$.</p>
<i>n</i>	The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>a</i> , <i>af</i> , <i>b</i>	<p>Arrays: <i>a</i>(size $\max(1, lda*n)$), <i>af</i>(size $\max(1, ldaf*n)$), <i>b</i> of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout.</p> <p>The array <i>a</i> contains the upper or the lower triangular part of the Hermitian matrix <i>A</i> (see <i>uplo</i>).</p> <p>The array <i>af</i> is an input argument if <i>fact</i> = 'F'. It contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> from the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ as computed by ?hetrf.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.</p>
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ipiv</i>	<p>Array, size at least $\max(1, n)$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. It contains details of the interchanges and the block structure of <i>D</i>, as determined by ?hetrf.</p> <p>If <i>ipiv</i>[<i>i</i>-1] = <i>k</i> > 0, then <i>d</i>_{<i>i</i>} is a 1-by-1 diagonal block, and the <i>i</i>-th row and column of <i>A</i> was interchanged with the <i>k</i>-th row and column.</p> <p>If <i>uplo</i> = 'U' and <i>ipiv</i>[<i>i</i>] = <i>ipiv</i>[<i>i</i>-1] = -<i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i>+1, and (<i>i</i>)-th row and column of <i>A</i> was interchanged with the <i>m</i>-th row and column.</p> <p>If <i>uplo</i> = 'L' and <i>ipiv</i>[<i>i</i>] = <i>ipiv</i>[<i>i</i>-1] = -<i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i>+1, and (<i>i</i>+1)-th row and column of <i>A</i> was interchanged with the <i>m</i>-th row and column.</p>
<i>ldx</i>	The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

<code>x</code>	Array, size $\max(1, \text{ldx} * \text{nrhs})$ for column major layout and $\max(1, \text{ldx} * n)$ for row major layout. If <code>info = 0</code> or <code>info = n+1</code> , the array <code>x</code> contains the solution matrix <code>X</code> to the system of equations.
<code>af, ipiv</code>	These arrays are output arguments if <code>fact = 'N'</code> . See the description of <code>af, ipiv</code> in <i>Input Arguments</i> section.
<code>rcond</code>	An estimate of the reciprocal condition number of the matrix <code>A</code> . If <code>rcond</code> is less than the machine precision (in particular, if <code>rcond = 0</code>), the matrix is singular to working precision. This condition is indicated by a return code of <code>info > 0</code> .
<code>ferr</code>	Array, size at least $\max(1, \text{nrhs})$. Contains the estimated forward error bound for each solution vector x_j (the j -th column of the solution matrix <code>X</code>). If <code>xtrue</code> is the true solution corresponding to x_j , <code>ferr[j-1]</code> is an estimated upper bound for the magnitude of the largest element in $(x_j) - x_{true}$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for <code>rcond</code> , and is almost always a slight overestimate of the true error.
<code>berr</code>	Array, size at least $\max(1, \text{nrhs})$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of <code>A</code> or <code>B</code> that makes x_j an exact solution.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, parameter `i` had an illegal value.

If `info = i`, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix `D` is exactly singular, so the solution and error bounds could not be computed; `rcond = 0` is returned.

If `info = i`, and $i = n + 1$, then `D` is nonsingular, but `rcond` is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of `rcond` would suggest.

See Also

[Matrix Storage Schemes](#)

?hesvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a Hermitian indefinite coefficient matrix `A` applying the diagonal pivoting factorization.

Syntax

```
lapack_int LAPACKE_chesvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, float* s, lapack_complex_float* b,
```

```
lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* rpvgrw,
float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp,
lapack_int nparams, const float* params );
```

```
lapack_int LAPACKE_zhesvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, double* s, lapack_complex_double* b,
lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double*
rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double*
err_bnds_comp, lapack_int nparams, const double* params );
```

Include Files

- mkl.h

Description

The routine uses the *diagonal pivoting* factorization to compute the solution to a complex/double complex system of linear equations $A \cdot X = B$, where A is an n -by- n Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine `?hesvxx` performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

$$\text{diag}(s) * A * \text{diag}(s) * \text{inv}(\text{diag}(s)) * X = \text{diag}(s) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and B by $\text{diag}(s) * B$.

2. If *fact* = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as

$$A = U * D * U^T, \text{ if } \text{uplo} = 'U',$$

$$\text{or } A = L * D * L^T, \text{ if } \text{uplo} = 'L',$$

where U or L is a product of permutation and unit upper (lower) triangular matrices, and D is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

3. If some $D(i, i) = 0$, so that D is exactly singular, the routine returns with *info* = i . Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless `params[0]` is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(r)$ so that it solves the original system before equilibration.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>fact</i>	<p>Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix <i>A</i> is supplied on entry, and if not, whether the matrix <i>A</i> should be equilibrated before it is factored.</p> <p>If <i>fact</i> = 'F', on entry, <i>af</i> and <i>ipiv</i> contain the factored form of <i>A</i>. If <i>equed</i> is not 'N', the matrix <i>A</i> has been equilibrated with scaling factors given by <i>s</i>. Parameters <i>a</i>, <i>af</i>, and <i>ipiv</i> are not modified.</p> <p>If <i>fact</i> = 'N', the matrix <i>A</i> will be copied to <i>af</i> and factored.</p> <p>If <i>fact</i> = 'E', the matrix <i>A</i> will be equilibrated, if necessary, copied to <i>af</i> and factored.</p>
<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of <i>A</i> is stored:</p> <p>If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored.</p> <p>If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.</p>
<i>n</i>	The number of linear equations; the order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; the number of columns of the matrices <i>B</i> and <i>X</i> ; $nrhs \geq 0$.
<i>a</i> , <i>af</i> , <i>b</i>	<p>Arrays: <i>a</i>(size $\max(lda*n)$), <i>af</i>(size $\max(ldaf*n)$), <i>b</i>, (size $\max(lb*nrhs)$ for column major layout and $\max(lb*n)$ for row major layout),.</p> <p>The array <i>a</i> contains the Hermitian matrix <i>A</i> as specified by <i>uplo</i>. If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> and the strictly upper triangular part of <i>a</i> is not referenced.</p> <p>The array <i>af</i> is an input argument if <i>fact</i> = 'F'. It contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> and <i>L</i> from the factorization $A = U^*D^*U^T$ or $A = L^*D^*L^T$ as computed by ?hetrf.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.</p>
<i>lda</i>	The leading dimension of the array <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	The leading dimension of the array <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. It contains details of the interchanges and the block structure of <i>D</i> as determined by ?sytrf .

If $ipiv[k-1] > 0$, rows and columns k and $ipiv[k-1]$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $uplo = 'U'$ and $ipiv[i] = ipiv[i - 1] = m < 0$, D has a 2-by-2 diagonal block in rows and columns i and $i + 1$, and the i -th row and column of A were interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv[i] = ipiv[i - 1] = m < 0$, D has a 2-by-2 diagonal block in rows and columns i and $i + 1$, and the $(i + 1)$ -st row and column of A were interchanged with the m -th row and column.

Must be 'N' or 'Y'.

equed is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

If $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$).

if $equed = 'Y'$, both row and column equilibration was done, that is, A has been replaced by $diag(s) * A * diag(s)$.

Array, size (n). The array s contains the scale factors for A . If $equed = 'Y'$, A is multiplied on the left and right by $diag(s)$.

This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.

If $fact = 'F'$ and $equed = 'Y'$, each element of s must be positive.

Each element of s should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

The leading dimension of the array b ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

The leading dimension of the output array x ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in the *Output Arguments* section below.

Specifies the number of parameters set in *params*. If ≤ 0 , the *params* array is never referenced and default values are used.

Array, size $\max(1, nparams)$. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

`params[0]` : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0 No refinement is performed and no error bounds are computed.

=1.0 Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

`params[1]` : Maximum number of residual computations allowed for refinement.

Default 10

Aggressive Set to 100 to permit convergence using approximate factorizations or factorizations other than LU . If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

`params[2]` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

Output Parameters

<code>x</code>	<p>Array, size $\max(1, \text{ldx} * \text{nrhs})$ for column major layout and $\max(1, \text{ldx} * n)$ for row major layout.</p> <p>If <code>info = 0</code>, the array <code>x</code> contains the solution n-by-<code>nrhs</code> matrix X to the original system of equations. Note that A and B are modified on exit if <code>equed</code> \neq 'N', and the solution to the equilibrated system is:</p> $\text{inv}(\text{diag}(s)) * X.$
<code>a</code>	If <code>fact = 'E'</code> and <code>equed = 'Y'</code> , overwritten by $\text{diag}(s) * A * \text{diag}(s)$.
<code>af</code>	If <code>fact = 'N'</code> , <code>af</code> is an output argument and on exit returns the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U * D * U^T$ or $A = L * D * L^T$.
<code>b</code>	<p>If <code>equed = 'N'</code>, B is not modified.</p> <p>If <code>equed = 'Y'</code>, B is overwritten by $\text{diag}(s) * B$.</p>
<code>s</code>	This array is an output argument if <code>fact</code> \neq 'F'. Each element of this array is a power of the radix. See the description of <code>s</code> in <i>Input Arguments</i> section.
<code>rcond</code>	Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If <code>rcond</code> is less than the machine precision, in particular, if <code>rcond = 0</code> , the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \leq n$, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

berr

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of *A* or *B* that makes x_j an exact solution.

err_bnds_norm

Array of size $nrhs * n_err_bnds$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for *chesvxx* and $\sqrt{n} * dlamch(\epsilon)$ for *zhesvxx*.

err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for *chesvxx* and $\sqrt{n} * dlamch(\epsilon)$ for *zhesvxx*. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for *chesvxx* and $\sqrt{n} * dlamch(\epsilon)$ for *zhesvxx* to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix *Z* are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * a$, where *s* scales each row by a power of the radix so all absolute row sums of *z* are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error err is stored in `err_bnds_norm[(err-1)*nrhs + i - 1]`.

`err_bnds_comp`

Array of size $nrhs * n_err_bnds$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params[2] = 0.0`), then `err_bnds_comp` is not accessed.

`err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for `chesvxx` and $\sqrt{n} * dlamch(\epsilon)$ for `zhesvxx`.

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for `chesvxx` and $\sqrt{n} * dlamch(\epsilon)$ for `zhesvxx`. This error bound should only be trusted if the previous boolean is true.

`err=3` Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for `chesvxx` and $\sqrt{n} * dlamch(\epsilon)$ for `zhesvxx` to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

The information for right-hand side i , where $1 \leq i \leq nrhs$, and type of error err is stored in `err_bnds_comp[(err-1)*nrhs + i - 1]`.

`ipiv`

If `fact = 'N'`, `ipiv` is an output argument and on exit contains details of the interchanges and the block structure D , as determined by `ssytrf` for single precision flavors and `dsytrf` for double precision flavors.

<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>params</i>	If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U_{\text{info}, \text{info}}$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n* + *j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params*[2] = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that for column major layout *err_bnds_norm*[*j* - 1] = 0.0 or *err_bnds_comp*[*j* - 1] = 0.0; or for row major layout *err_bnds_norm*[(*j* - 1) * *n_err_bnds*] = 0.0 or *err_bnds_comp*[(*j* - 1) * *n_err_bnds*] = 0.0). See the definition of *err_bnds_norm* and *err_bnds_comp* for *err* = 1. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

Matrix Storage Schemes

?spsv

Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A stored in packed format, and multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_sspsv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , float * ap , lapack_int * ipiv , float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_dspsv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , double * ap , lapack_int * ipiv , double * b , lapack_int ldb );
```

```
lapack_int LAPACKE_cspsv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , lapack_complex_float * ap , lapack_int * ipiv , lapack_complex_float * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_zspsv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , lapack_complex_double * ap , lapack_int * ipiv , lapack_complex_double * b ,
lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for X the real or complex system of linear equations $A * X = B$, where A is an n -by- n symmetric matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U * D * U^T$ or $A = L * D * L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A * X = B$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>ap, b</i>	Arrays: <i>ap</i> (size $\max(1, n*(n+1)/2)$, <i>bof</i> size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout). The array <i>ap</i> contains the factor U or L , as specified by <i>uplo</i> , in <i>packed storage</i> (see Matrix Storage Schemes). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>ap</i>	The block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by ?spturf , stored as a packed triangular matrix in the same storage format as A .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix X .
<i>ipiv</i>	Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D , as determined by ?spturf . If <i>ipiv</i> [<i>i</i> -1] = $k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column. If <i>uplo</i> = 'U' and <i>ipiv</i> [<i>i</i>] = <i>ipiv</i> [<i>i</i> -1] = $-m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and i -th row and column of A was interchanged with the m -th row and column. If <i>uplo</i> = 'L' and <i>ipiv</i> [<i>i</i> -1] = <i>ipiv</i> [<i>i</i>] = $-m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular, so the solution could not be computed.

See Also

Matrix Storage Schemes

?spsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric coefficient matrix A stored in packed format, and provides error bounds on the solution.

Syntax

```

lapack_int LAPACKE_sspsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const float* ap, float* afp, lapack_int* ipiv, const float* b,
lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_dspsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const double* ap, double* afp, lapack_int* ipiv, const double* b,
lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

lapack_int LAPACKE_cspsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float* ap, lapack_complex_float* afp, lapack_int*
ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zspsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* ap, lapack_complex_double* afp,
lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

```

Include Files

- mkl.h

Description

The routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where *A* is a *n*-by-*n* symmetric matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?spsvx performs the following steps:

1. If *fact* = 'N', the diagonal pivoting method is used to factor the matrix *A*. The form of the factorization is $A = U \cdot D \cdot U^T$ or $A = L \cdot D \cdot L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>fact</i>	<p>Must be 'F' or 'N'.</p> <p>Specifies whether or not the factored form of the matrix A has been supplied on entry.</p> <p>If <i>fact</i> = 'F': on entry, <i>afp</i> and <i>ipiv</i> contain the factored form of A. Arrays <i>ap</i>, <i>afp</i>, and <i>ipiv</i> are not modified.</p> <p>If <i>fact</i> = 'N', the matrix A is copied to <i>afp</i> and factored.</p>
<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>Indicates whether the upper or lower triangular part of A is stored and how A is factored:</p> <p>If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the symmetric matrix A, and A is factored as U^*D*U^T.</p> <p>If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the symmetric matrix A; A is factored as L^*D*L^T.</p>
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>ap</i> , <i>afp</i> , <i>b</i>	<p>Arrays: <i>ap</i> (size $\max(1, n*(n+1)/2)$), <i>afp</i> (size $\max(1, n*(n+1)/2)$), <i>bof</i> (size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout).</p> <p>The array <i>ap</i> contains the upper or lower triangle of the symmetric matrix A in <i>packed storage</i> (see Matrix Storage Schemes).</p> <p>The array <i>afp</i> is an input argument if <i>fact</i> = 'F'. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U^*D*U^T$ or $A = L^*D*L^T$ as computed by ?spturf, in the same storage format as A.</p> <p>The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations.</p>
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
<i>ipiv</i>	Array, size at least $\max(1, n)$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. It contains details of the interchanges and the block structure of D , as determined by ?spturf .

If $ipiv[i-1] = k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv[i]=ipiv[i-1] = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and i -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv[i-1]=ipiv[i] = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

ldx

The leading dimension of the output array x ; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

x

Array, size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout.

If $info = 0$ or $info = n+1$, the array x contains the solution matrix X to the system of equations.

afp, ipiv

These arrays are output arguments if $fact = 'N'$. See the description of *afp, ipiv* in *Input Arguments* section.

rcond

An estimate of the reciprocal condition number of the matrix A . If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr, berr

Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

Return Values

This function returns a value *info*.

If $info = 0$, the execution is successful.

If $info = -i$, parameter i had an illegal value.

If $info = i$, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then D is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

See Also

Matrix Storage Schemes

?hpsv

Computes the solution to the system of linear equations with a Hermitian coefficient matrix A stored in packed format, and multiple right-hand sides.

Syntax

```
lapack_int LAPACKE_chpsv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , lapack_complex_float * ap , lapack_int * ipiv , lapack_complex_float * b ,
lapack_int ldb );
```

```
lapack_int LAPACKE_zhpsv (int matrix_layout , char uplo , lapack_int n , lapack_int
nrhs , lapack_complex_double * ap , lapack_int * ipiv , lapack_complex_double * b ,
lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for X the system of linear equations $A^*X = B$, where A is an n -by- n Hermitian matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U^*D^*U^H$ or $A = L^*D^*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	The number of right-hand sides; the number of columns in B ; $nrhs \geq 0$.
<i>ap, b</i>	Arrays: <i>ap</i> (size $\max(1, n*(n+1)/2)$, <i>bof</i> size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout). The array <i>ap</i> contains the factor U or L , as specified by <i>uplo</i> , in <i>packed storage</i> (see Matrix Storage Schemes). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations.
<i>ldb</i>	The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

<i>ap</i>	The block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by ?hptrf , stored as a packed triangular matrix in the same storage format as A .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix X .
<i>ipiv</i>	<p>Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D, as determined by ?hptrf.</p> <p>If <i>ipiv</i>[<i>i</i>-1] = <i>k</i> > 0, then d_{ii} is a 1-by-1 block, and the <i>i</i>-th row and column of A was interchanged with the <i>k</i>-th row and column.</p> <p>If <i>uplo</i> = 'U' and <i>ipiv</i>[<i>i</i>]=<i>ipiv</i>[<i>i</i>-1] = -<i>m</i> < 0, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i>+1, and <i>i</i>-th row and column of A was interchanged with the <i>m</i>-th row and column.</p> <p>If <i>uplo</i> = 'L' and <i>ipiv</i>[<i>i</i>-1]=<i>ipiv</i>[<i>i</i>] = -<i>m</i> < 0, then D has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i>+1, and (<i>i</i>+1)-th row and column of A was interchanged with the <i>m</i>-th row and column.</p>

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

See Also

Matrix Storage Schemes

?hpsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a Hermitian coefficient matrix A stored in packed format, and provides error bounds on the solution.

Syntax

```
lapack_int LAPACKE_chpsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float* ap, lapack_complex_float* afp, lapack_int*
ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zhpsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* ap, lapack_complex_double* afp,
lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A \cdot X = B$, where A is a n -by- n Hermitian matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?hpsvx` performs the following steps:

1. If `fact = 'N'`, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U \cdot D \cdot U^H$ or $A = L \cdot D \cdot L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with `info = i`. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, `info = n+1` is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>fact</code>	Must be 'F' or 'N'. Specifies whether or not the factored form of the matrix A has been supplied on entry. If <code>fact = 'F'</code> : on entry, <code>afp</code> and <code>ipiv</code> contain the factored form of A . Arrays <code>ap</code> , <code>afp</code> , and <code>ipiv</code> are not modified. If <code>fact = 'N'</code> , the matrix A is copied to <code>afp</code> and factored.
<code>uplo</code>	Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <code>uplo = 'U'</code> , the array <code>ap</code> stores the upper triangular part of the Hermitian matrix A , and A is factored as $U \cdot D \cdot U^H$. If <code>uplo = 'L'</code> , the array <code>ap</code> stores the lower triangular part of the Hermitian matrix A , and A is factored as $L \cdot D \cdot L^H$.
<code>n</code>	The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<code>ap, afp, b</code>	Arrays: <code>ap</code> (size $\max(1, n \cdot (n+1)/2)$), <code>afp</code> (size $\max(1, n \cdot (n+1)/2)$), <code>bof</code> (size $\max(1, ldb \cdot nrhs)$) for column major layout and $\max(1, ldb \cdot n)$ for row major layout. The array <code>ap</code> contains the upper or lower triangle of the Hermitian matrix A in <i>packed storage</i> (see Matrix Storage Schemes).

The array *afp* is an input argument if *fact* = 'F'. It contains the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L* from the factorization $A = U * D * U^H$ or $A = L * D * L^H$ as computed by `?hptrf`, in the same storage format as *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

ldb

The leading dimension of *b*; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ipiv

Array, size at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by `?hptrf`.

If $ipiv[i-1] = k > 0$, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If $uplo = 'U'$ and $ipiv[i]=ipiv[i-1] = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i+1*, and *i*-th row and column of *A* was interchanged with the *m*-th row and column.

If $uplo = 'L'$ and $ipiv[i-1]=ipiv[i] = -m < 0$, then *D* has a 2-by-2 block in rows/columns *i* and *i+1*, and (*i+1*)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx

The leading dimension of the output array *x*; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

x

Array, size $\max(1, ldx * nrhs)$ for column major layout and $\max(1, ldx * n)$ for row major layout.

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations.

afp, ipiv

These arrays are output arguments if *fact* = 'N'. See the description of *afp, ipiv* in *Input Arguments* section.

rcond

An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector x_j (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to x_j , $ferr[j-1]$ is an estimated upper bound for the magnitude of the largest element in $(x_j - xtrue)$ divided by the magnitude of the largest element in x_j . The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector x_j , that is, the smallest relative change in any element of *A* or *B* that makes x_j an exact solution.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, parameter *i* had an illegal value.

If *info* = *i*, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *i*, and $i = n + 1$, then *D* is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

See Also

[Matrix Storage Schemes](#)

LAPACK Least Squares and Eigenvalue Problem Routines

This section includes descriptions of LAPACK [computational routines](#) and [driver routines](#) for solving linear least squares problems, eigenvalue and singular value problems, and performing a number of related computational tasks. For a full reference on LAPACK routines and related information see [\[LUG\]](#).

Least Squares Problems. A typical *least squares problem* is as follows: given a matrix *A* and a vector *b*, find the vector *x* that minimizes the sum of squares $\sum_i ((Ax)_i - b_i)^2$ or, equivalently, find the vector *x* that minimizes the 2-norm $\|Ax - b\|_2$.

In the most usual case, *A* is an *m*-by-*n* matrix with $m \geq n$ and $\text{rank}(A) = n$. This problem is also referred to as finding the *least squares solution* to an *overdetermined* system of linear equations (here we have more equations than unknowns). To solve this problem, you can use the QR factorization of the matrix *A* (see [QR Factorization](#)).

If $m < n$ and $\text{rank}(A) = m$, there exist an infinite number of solutions *x* which exactly satisfy $Ax = b$, and thus minimize the norm $\|Ax - b\|_2$. In this case it is often useful to find the unique solution that minimizes $\|x\|_2$. This problem is referred to as finding the *minimum-norm solution* to an *underdetermined* system of linear equations (here we have more unknowns than equations). To solve this problem, you can use the LQ factorization of the matrix *A* (see [LQ Factorization](#)).

In the general case you may have a *rank-deficient least squares problem*, with $\text{rank}(A) < \min(m, n)$: find the *minimum-norm least squares solution* that minimizes both $\|x\|_2$ and $\|Ax - b\|^2$. In this case (or when the rank of *A* is in doubt) you can use the QR factorization with pivoting or *singular value decomposition* (see [Singular Value Decomposition](#)).

Eigenvalue Problems. The eigenvalue problems (from German *eigen* "own") are stated as follows: given a matrix *A*, find the *eigenvalues* λ and the corresponding *eigenvectors* *z* that satisfy the equation

$$Az = \lambda z \text{ (right eigenvectors } z)$$

or the equation

$$z^H A = \lambda z^H \text{ (left eigenvectors } z).$$

If *A* is a real symmetric or complex Hermitian matrix, the above two equations are equivalent, and the problem is called a *symmetric eigenvalue problem*. Routines for solving this type of problems are described in the topic [Symmetric Eigenvalue Problems](#).

Routines for solving eigenvalue problems with nonsymmetric or non-Hermitian matrices are described in the topic [Nonsymmetric Eigenvalue Problems](#).

The library also includes routines that handle *generalized symmetric-definite eigenvalue problems*: find the eigenvalues λ and the corresponding eigenvectors *x* that satisfy one of the following equations:

$$Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z,$$

where A is symmetric or Hermitian, and B is symmetric positive-definite or Hermitian positive-definite. Routines for reducing these problems to standard symmetric eigenvalue problems are described in the topic [Generalized Symmetric-Definite Eigenvalue Problems](#).

To solve a particular problem, you usually call several computational routines. Sometimes you need to combine the routines of this chapter with other LAPACK routines described in "LAPACK Routines: Linear Equations" as well as with BLAS routines described in "BLAS and Sparse BLAS Routines".

For example, to solve a set of least squares problems minimizing $\|Ax - b\|^2$ for all columns b of a given matrix B (where A and B are real matrices), you can call `?geqrf` to form the factorization $A = QR$, then call `?ormqr` to compute $C = Q^H B$ and finally call the BLAS routine `?trsm` to solve for X the system of equations $RX = C$.

Another way is to call an appropriate driver routine that performs several tasks in one call. For example, to solve the least squares problem the driver routine `?gels` can be used.

LAPACK Least Squares and Eigenvalue Problem Computational Routines

In the topics that follow, the descriptions of LAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

[Orthogonal Factorizations](#)

[Singular Value Decomposition](#)

[Symmetric Eigenvalue Problems](#)

[Generalized Symmetric-Definite Eigenvalue Problems](#)

[Nonsymmetric Eigenvalue Problems](#)

[Generalized Nonsymmetric Eigenvalue Problems](#)

[Generalized Singular Value Decomposition](#)

See also the respective [driver routines](#).

Orthogonal Factorizations: LAPACK Computational Routines

This topic describes the LAPACK routines for the QR (RQ) and LQ (QL) factorization of matrices. Routines for the RZ factorization as well as for generalized QR and RQ factorizations are also included.

QR Factorization. Assume that A is an m -by- n matrix to be factored.

If $m \geq n$, the QR factorization is given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = (Q_1, Q_2) \begin{pmatrix} R \\ 0 \end{pmatrix}$$

where R is an n -by- n upper triangular matrix with real diagonal elements, and Q is an m -by- m orthogonal (or unitary) matrix.

You can use the QR factorization for solving the following least squares problem: minimize $\|Ax - b\|^2$ where A is a full-rank m -by- n matrix ($m \geq n$). After factoring the matrix, compute the solution x by solving $Rx = (Q_1)^T b$.

If $m < n$, the QR factorization is given by

$$A = QR = Q(R_1 R_2)$$

where R is trapezoidal, R_1 is upper triangular and R_2 is rectangular.

Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

LQ Factorization LQ factorization of an m -by- n matrix A is as follows. If $m \leq n$,

$$A = \begin{pmatrix} L & 0 \end{pmatrix} Q = \begin{pmatrix} L & 0 \end{pmatrix} \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = \begin{pmatrix} LQ_1 \end{pmatrix}$$

where L is an m -by- m lower triangular matrix with real diagonal elements, and Q is an n -by- n orthogonal (or unitary) matrix.

If $m > n$, the LQ factorization is

$$A = \begin{pmatrix} L_1 \\ L_2 \end{pmatrix} Q$$

where L_1 is an n -by- n lower triangular matrix, L_2 is rectangular, and Q is an n -by- n orthogonal (or unitary) matrix.

You can use the LQ factorization to find the minimum-norm solution of an underdetermined system of linear equations $Ax = b$ where A is an m -by- n matrix of rank m ($m < n$). After factoring the matrix, compute the solution vector x as follows: solve $L_1 y = b$ for y , and then compute $x = (Q_1)^H y$.

Table "Computational Routines for Orthogonal Factorization" lists LAPACK routines that perform orthogonal factorization of matrices.

Computational Routines for Orthogonal Factorization

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	geqrf	geqpf	orgqr	ormqr
	geqrfp	geqp3	ungqr	unmqr
general matrices, blocked QR factorization	geqrt			gemqrt
general matrices, RQ factorization	gerqf		orgrq	ormrq
			ungrq	unmrq

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, LQ factorization	gelqf		orglq unglq	ormlq unmlq
general matrices, QL factorization	geqlf		orgql ungql	ormql unmql
trapezoidal matrices, RZ factorization	tzzrf			ormrz unmrz
pair of matrices, generalized QR factorization	ggqrf			
pair of matrices, generalized RQ factorization	ggrqf			
triangular-pentagonal matrices, blocked QR factorization	tpqrt			tpmqrt

?geqrf

Computes the QR factorization of a general m -by- n matrix.

Syntax

```
lapack_int LAPACKE_sgeqrf (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, float* tau);

lapack_int LAPACKE_dgeqrf (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, double* tau);

lapack_int LAPACKE_cgeqrf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);

lapack_int LAPACKE_zgeqrf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- `mkl.h`

Description

The routine forms the QR factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>m</code>	The number of rows in the matrix A ($m \geq 0$).
<code>n</code>	The number of columns in A ($n \geq 0$).
<code>a</code>	Array a of size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout contains the matrix A .
<code>lda</code>	The leading dimension of a ; at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.

Output Parameters

<code>a</code>	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the $\min(m,n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array <code>tau</code> , present the orthogonal matrix Q as a product of $\min(m,n)$ elementary reflectors (see Orthogonal Factorizations).
<code>tau</code>	Array, size at least $\max(1, \min(m, n))$. Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors (see Orthogonal Factorizations).

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$\begin{aligned} (4/3)n^3 & \quad \text{if } m = n, \\ (2/3)n^2(3m-n) & \quad \text{if } m > n, \\ (2/3)m^2(3n-m) & \quad \text{if } m < n. \end{aligned}$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

<code>?geqrf</code> (this routine)	to factorize $A = QR$;
<code>ormqr</code>	to compute $C = Q^T*B$ (for real matrices);
<code>unmqr</code>	to compute $C = Q^H*B$ (for complex matrices);

`trsm` (a BLAS routine) to solve $R^*X = C$.

(The columns of the computed X are the least squares solution vectors x .)

To compute the elements of Q explicitly, call

`orgqr` (for real matrices)

`ungqr` (for complex matrices).

See Also

[mkl_progress](#)

Matrix Storage Schemes

?geqrfp

Computes the QR factorization of a general m -by- n matrix with non-negative diagonal elements.

Syntax

```
lapack_int LAPACKE_sgeqrfp (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, float* tau);

lapack_int LAPACKE_dgeqrfp (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, double* tau);

lapack_int LAPACKE_cgeqrfp (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);

lapack_int LAPACKE_zgeqrfp (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- `mkl.h`

Description

The routine forms the QR factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed. The diagonal entries of R are real and nonnegative.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>m</i>	The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	The number of columns in A ($n \geq 0$).

<i>a</i>	Array, size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout, containing the matrix <i>A</i> .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the $\min(m, n)$ -by- <i>n</i> upper trapezoidal matrix <i>R</i> (<i>R</i> is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i> , present the orthogonal matrix <i>Q</i> as a product of $\min(m, n)$ elementary reflectors (see Orthogonal Factorizations). The diagonal elements of the matrix <i>R</i> are real and non-negative.
<i>tau</i>	Array, size at least $\max(1, \min(m, n))$. Contains scalars that define elementary reflectors for the matrix <i>Q</i> in its decomposition in a product of elementary reflectors (see Orthogonal Factorizations).

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$\begin{aligned} (4/3)n^3 & \quad \text{if } m = n, \\ (2/3)n^2(3m-n) & \quad \text{if } m > n, \\ (2/3)m^2(3n-m) & \quad \text{if } m < n. \end{aligned}$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A*x - b\|_2$ for all columns *b* of a given matrix *B*, you can call the following:

<code>?geqrfp</code> (this routine)	to factorize $A = QR$;
<code>ormqr</code>	to compute $C = Q^T*B$ (for real matrices);
<code>unmqr</code>	to compute $C = Q^H*B$ (for complex matrices);
<code>trsm</code> (a BLAS routine)	to solve $R*X = C$.

(The columns of the computed *X* are the least squares solution vectors *x*.)

To compute the elements of *Q* explicitly, call

`orgqr` (for real matrices)

[ungqr](#) (for complex matrices).

See Also

[mkl_progress](#)

Matrix Storage Schemes

?geqrt

Computes a blocked QR factorization of a general real or complex matrix using the compact WY representation of Q.

Syntax

```
lapack_int LAPACKE_sgeqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int nb, float* a, lapack_int lda, float* t, lapack_int ldt);
```

```
lapack_int LAPACKE_dgeqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int nb, double* a, lapack_int lda, double* t, lapack_int ldt);
```

```
lapack_int LAPACKE_cgeqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int nb, lapack_complex_float* a, lapack_int lda, lapack_complex_float* t, lapack_int ldt);
```

```
lapack_int LAPACKE_zgeqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int nb, lapack_complex_double* a, lapack_int lda, lapack_complex_double* t, lapack_int ldt);
```

Include Files

- `mkl.h`

Description

The strictly lower triangular matrix V contains the elementary reflectors $H(i)$ in the i th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix V is

$$V = \begin{bmatrix} 1 & & & & \\ v_1 & 1 & & & \\ v_1 & v_2 & 1 & & \\ v_1 & v_2 & v_3 & & \\ v_1 & v_2 & v_3 & & \end{bmatrix}$$

where v_i represents one of the vectors that define $H(i)$. The vectors are returned in the lower triangular part of array a .

NOTE

The 1s along the diagonal of V are not stored in a .

Let $k = \min(m, n)$. The number of blocks is $b = \text{ceiling}(k/nb)$, where each block is of order nb except for the last block, which is of order $ib = k - (b-1)*nb$. For each of the b blocks, a upper triangular block reflector factor is computed: $t1, t2, \dots, tb$. The nb -by- nb (and ib -by- ib for the last block) ts are stored in the nb -by- n array t as

$t = (t1t2 \dots tb)$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	The number of columns in A ($n \geq 0$).
<i>nb</i>	The block size to be used in the blocked QR ($\min(m, n) \geq nb \geq 1$).
<i>a</i>	Array a of size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout contains the m -by- n matrix A .
<i>lda</i>	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
<i>ldt</i>	The leading dimension of t ; at least nb for column major layout and $\max(1, \min(m, n))$ for row major layout.

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the $\min(m, n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array t , present the orthogonal matrix Q as a product of $\min(m, n)$ elementary reflectors (see Orthogonal Factorizations).
<i>t</i>	Array, size $\max(1, ldt*\min(m, n))$ for column major layout and $\max(1, ldt*nb)$ for row major layout. The upper triangular block reflector's factors stored as a sequence of upper triangular blocks.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* < 0 and *info* = $-i$, the i -th parameter had an illegal value.

?gemqrt

Multiplies a general matrix by the orthogonal/unitary matrix Q of the QR factorization formed by ?geqrt.

Syntax

```
lapack_int LAPACKE_sgemqrt (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int nb, const float* v, lapack_int ldv, const float*
t, lapack_int ldt, float* c, lapack_int ldc);
```

```
lapack_int LAPACKE_dgemqrt (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int nb, const double* v, lapack_int ldv, const
double* t, lapack_int ldt, double* c, lapack_int ldc);
```

```
lapack_int LAPACKE_cgemqrt (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int nb, const lapack_complex_float* v, lapack_int
ldv, const lapack_complex_float* t, lapack_int ldt, lapack_complex_float* c, lapack_int
ldc);
```

```
lapack_int LAPACKE_zgemqrt (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int nb, const lapack_complex_double* v, lapack_int
ldv, const lapack_complex_double* t, lapack_int ldt, lapack_complex_double* c,
lapack_int ldc);
```

Include Files

- mkl.h

Description

The ?gemqrt routine overwrites the general real or complex m -by- n matrix C with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	Q^*C	C^*Q
$trans = 'T':$	$Q^T C$	$C^* Q^T$
$trans = 'C':$	$Q^H C$	$C^* Q^H$

where Q is a real orthogonal (complex unitary) matrix defined as the product of k elementary reflectors

$Q = H(1) H(2) \dots H(k) = I - V^* T^* V^T$ for real flavors, and

$Q = H(1) H(2) \dots H(k) = I - V^* T^* V^H$ for complex flavors,

generated using the compact WY representation as returned by [geqrt](#). Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	='L': apply Q , Q^T , or Q^H from the left. ='R': apply Q , Q^T , or Q^H from the right.
<i>trans</i>	='N', no transpose, apply Q . ='T', transpose, apply Q^T . ='C', transpose, apply Q^H .
<i>m</i>	The number of rows in the matrix C , ($m \geq 0$).

n	The number of columns in the matrix C , ($n \geq 0$).
k	The number of elementary reflectors whose product defines the matrix Q . Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.
nb	The block size used for the storage of t , $k \geq nb \geq 1$. This must be the same value of nb used to generate t in geqrt .
v	Array of size $\max(1, ldv*k)$ for column major layout, $\max(1, ldv*m)$ for row major layout and $side = 'L'$, and $\max(1, ldv*n)$ for row major layout and $side = 'R'$. The i th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by geqrt in the first k columns of its array argument a .
ldv	The leading dimension of the array v . if $side = 'L'$, ldv must be at least $\max(1, m)$ for column major layout and $\max(1, k)$ for row major layout; if $side = 'R'$, ldv must be at least $\max(1, n)$ for column major layout and $\max(1, k)$ for row major layout.
t	Array, size $\max(1, ldt*\min(m, n))$ for column major layout and $\max(1, ldt*nb)$ for row major layout. The upper triangular factors of the block reflectors as returned by geqrt .
ldt	The leading dimension of the array t . ldt must be at least nb for column major layout and $\max(1, k)$ for row major layout.
c	The m -by- n matrix C .
ldc	The leading dimension of the array c . ldc must be at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

c	Overwritten by the product $Q*C$, $C*Q$, Q^T*C , $C*Q^T$, Q^H*C , or $C*Q^H$ as specified by $side$ and $trans$.
-----	--

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

?geqpf

Computes the QR factorization of a general m -by- n matrix with pivoting.

Syntax

```
lapack_int LAPACKE_sgeqpf (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, lapack_int* jpvt, float* tau);

lapack_int LAPACKE_dgeqpf (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, lapack_int* jpvt, double* tau);

lapack_int LAPACKE_cgeqpf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_int* jpvt, lapack_complex_float* tau);

lapack_int LAPACKE_zgeqpf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_int* jpvt, lapack_complex_double*
tau);
```

Include Files

- mkl.h

Description

The routine is deprecated and has been replaced by routine [geqp3](#).

The routine `?geqpf` forms the QR factorization of a general m -by- n matrix A with column pivoting: $A*P = Q*R$ (see [Orthogonal Factorizations](#)). Here P denotes an n -by- n permutation matrix.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	The number of columns in A ($n \geq 0$).
<i>a</i>	Array <i>a</i> of size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout contains the matrix A .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
<i>jpvt</i>	<p>Array, size at least $\max(1, n)$.</p> <p>On entry, if $jpvt[i - 1] > 0$, the i-th column of A is moved to the beginning of $A*P$ before the computation, and fixed in place during the computation.</p> <p>If $jpvt[i - 1] = 0$, the ith column of A is a free column (that is, it may be interchanged during the computation with any other free column).</p>

Output Parameters

<i>a</i>	<p>Overwritten by the factorization data as follows:</p> <p>The elements on and above the diagonal of the array contain the $\min(m, n)$-by-n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i>, present the orthogonal matrix Q as a product of $\min(m, n)$ elementary reflectors (see Orthogonal Factorizations).</p>
----------	--

<code>tau</code>	Array, size at least $\max(1, \min(m, n))$. Contains additional information on the matrix Q .
<code>jpvt</code>	Overwritten by details of the permutation matrix P in the factorization $A^*P = Q^*R$. More precisely, the columns of A^*P are the columns of A in the following order: <code>jpvt[0], jpvt[1], ..., jpvt[n - 1]</code> .

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$\begin{aligned} (4/3)n^3 & \quad \text{if } m = n, \\ (2/3)n^2(3m-n) & \quad \text{if } m > n, \\ (2/3)m^2(3n-m) & \quad \text{if } m < n. \end{aligned}$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

<code>?geqp3</code> (this routine)	to factorize $A^*P = Q^*R$;
<code>ormqr</code>	to compute $C = Q^T * B$ (for real matrices);
<code>unmqr</code>	to compute $C = Q^H * B$ (for complex matrices);
<code>trsm</code> (a BLAS routine)	to solve $R^*X = C$.

(The columns of the computed X are the permuted least squares solution vectors x ; the output array `jpvt` specifies the permutation order.)

To compute the elements of Q explicitly, call

<code>orgqr</code>	(for real matrices)
<code>ungqr</code>	(for complex matrices).

?geqp3

Computes the QR factorization of a general m -by- n matrix with column pivoting using level 3 BLAS.

Syntax

```
lapack_int LAPACKE_sgeqp3 (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, lapack_int* jpvt, float* tau);
```

```

lapack_int LAPACKE_dgeqp3 (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, lapack_int* jpvt, double* tau);

lapack_int LAPACKE_cgeqp3 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_int* jpvt, lapack_complex_float* tau);

lapack_int LAPACKE_zgeqp3 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_int* jpvt, lapack_complex_double*
tau);

```

Include Files

- mkl.h

Description

The routine forms the QR factorization of a general m -by- n matrix A with column pivoting: $A^*P = Q^*R$ (see [Orthogonal Factorizations](#)) using Level 3 BLAS. Here P denotes an n -by- n permutation matrix. Use this routine instead of [geqpf](#) for better performance.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	The number of columns in A ($n \geq 0$).
<i>a</i>	Array <i>a</i> of size $\max(1, lda \cdot n)$ for column major layout and $\max(1, lda \cdot m)$ for row major layout contains the matrix A .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
<i>jpvt</i>	<p>Array, size at least $\max(1, n)$.</p> <p>On entry, if $jpvt[i - 1] \neq 0$, the i-th column of A is moved to the beginning of AP before the computation, and fixed in place during the computation.</p> <p>If $jpvt[i - 1] = 0$, the i-th column of A is a free column (that is, it may be interchanged during the computation with any other free column).</p>

Output Parameters

<i>a</i>	<p>Overwritten by the factorization data as follows:</p> <p>The elements on and above the diagonal of the array contain the $\min(m, n)$-by-n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i>, present the orthogonal matrix Q as a product of $\min(m, n)$ elementary reflectors (see Orthogonal Factorizations).</p>
<i>tau</i>	Array, size at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors for the matrix Q .

`jpvt`

Overwritten by details of the permutation matrix P in the factorization $A^*P = Q^*R$. More precisely, the columns of AP are the columns of A in the following order:

`jpvt[0], jpvt[1], ..., jpvt[n - 1]`.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

<code>?geqp3</code> (this routine)	to factorize $A^*P = Q^*R$;
<code>ormqr</code>	to compute $C = Q^T*B$ (for real matrices);
<code>unmqr</code>	to compute $C = Q^H*B$ (for complex matrices);
<code>trsm</code> (a BLAS routine)	to solve $R^*X = C$.

(The columns of the computed X are the permuted least squares solution vectors x ; the output array `jpvt` specifies the permutation order.)

To compute the elements of Q explicitly, call

<code>orgqr</code>	(for real matrices)
<code>ungqr</code>	(for complex matrices).

`?orgqr`

Generates the real orthogonal matrix Q of the QR factorization formed by `?geqrf`.

Syntax

```
lapack_int LAPACKE_sorgqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
float* a, lapack_int lda, const float* tau);
```

```
lapack_int LAPACKE_dorgqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
double* a, lapack_int lda, const double* tau);
```

Include Files

- `mkl.h`

Description

The routine generates the whole or part of m -by- m orthogonal matrix Q of the QR factorization formed by the routine `?geqrf` or `geqpf`. Use this routine after a call to `sgeqrf/dgeqrf` or `sgeqpf/dgeqpf`.

Usually Q is determined from the QR factorization of an m by p matrix A with $m \geq p$. To compute the whole matrix Q , use:

```
LAPACKE_?orgqr(matrix_layout, m, m, p, a, lda, tau)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
LAPACKE_?orgqr(matrix_layout, m, p, p, a, lda)
```

To compute the matrix Q^k of the QR factorization of leading k columns of the matrix A :

```
LAPACKE_?orgqr(matrix_layout, m, m, k, a, lda, tau)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by leading k columns of the matrix A):

```
LAPACKE_?orgqr(matrix_layout, m, k, k, a, lda, tau)
```

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The order of the orthogonal matrix Q ($m \geq 0$).
<i>n</i>	The number of columns of Q to be computed ($0 \leq n \leq m$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).
<i>a, tau</i>	Arrays: <i>a</i> and <i>tau</i> are the arrays returned by <code>sgeqrf / dgeqrf</code> or <code>sgeqpf / dgeqpf</code> . The size of <i>a</i> is $\max(1, lda * n)$ for column major layout and $\max(1, lda * m)$ for row major layout . The size of <i>tau</i> must be at least $\max(1, k)$.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	Overwritten by n leading columns of the m -by- m orthogonal matrix Q .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed Q differs from an exactly orthogonal matrix by a matrix E such that

$\|E\|_2 = O(\varepsilon) \|A\|_2$ where ε is the machine precision.

The total number of floating-point operations is approximately $4 * m * n * k - 2 * (m + n) * k^2 + (4/3) * k^3$.

If $n = k$, the number is approximately $(2/3) * n^2 * (3m - n)$.

The complex counterpart of this routine is [ungqr](#).

`?ormqr`

Multiplies a real matrix by the orthogonal matrix Q of the QR factorization formed by `?geqrf` or `?geqpf`.

Syntax

```
lapack_int LAPACKE_sormqr (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const float* a, lapack_int lda, const float* tau, float* c,
lapack_int ldc);
```

```
lapack_int LAPACKE_dormqr (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const double* a, lapack_int lda, const double* tau, double*
c, lapack_int ldc);
```

Include Files

- `mkl.h`

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the QR factorization formed by the routine [?geqrf](#) or [?geqpf](#).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products Q^*C , $Q^T * C$, $C * Q$, or $C * Q^T$ (overwriting the result on C).

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>side</code>	Must be either 'L' or 'R'. If <code>side='L'</code> , Q or Q^T is applied to C from the left. If <code>side='R'</code> , Q or Q^T is applied to C from the right.
<code>trans</code>	Must be either 'N' or 'T'. If <code>trans='N'</code> , the routine multiplies C by Q . If <code>trans='T'</code> , the routine multiplies C by Q^T .
<code>m</code>	The number of rows in the matrix C ($m \geq 0$).
<code>n</code>	The number of columns in C ($n \geq 0$).
<code>k</code>	The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <code>side='L'</code> ; $0 \leq k \leq n$ if <code>side='R'</code> .
<code>a, tau, c</code>	Arrays: a and tau are the arrays returned by <code>sgeqrf</code> / <code>dgeqrf</code> or <code>sgeqpf</code> / <code>dgeqpf</code> .

The size of a is $\max(1, lda*k)$ for column major layout, $\max(1, lda*m)$ for row major layout and $side = 'L'$, and $\max(1, lda*n)$ for row major layout and $side = 'R'$.

The size of τ must be at least $\max(1, k)$.

Array c of size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout contains the m -by- n matrix C .

lda

The leading dimension of a . Constraints:

if $side = 'L'$, $lda \geq \max(1, m)$ for column major layout and $\max(1, k)$ for row major layout ;

if $side = 'R'$, $lda \geq \max(1, n)$ for column major layout and $\max(1, k)$ for row major layout.

ldc

The leading dimension of c . Constraint:

$ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

c

Overwritten by the product Q^*C , $Q^T C$, C^*Q , or C^*Q^T (as specified by $side$ and $trans$).

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is [unmqr](#).

?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by ?geqrf.

Syntax

```
lapack_int LAPACKE_cungqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);
```

```
lapack_int LAPACKE_zungqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);
```

Include Files

- `mkl.h`

Description

The routine generates the whole or part of m -by- m unitary matrix Q of the QR factorization formed by the routines [?geqrf](#) or [geqpf](#). Use this routine after a call to [cgeqrf/zgeqrf](#) or [cgeqpf/zgeqpf](#).

Usually Q is determined from the QR factorization of an m by p matrix A with $m \geq p$. To compute the whole matrix Q , use:

```
LAPACKE_?ungqr(matrix_layout, m, m, p, a, lda, tau)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
LAPACKE_?ungqr(matrix_layout, m, p, p, a, lda, tau)
```

To compute the matrix Q^k of the QR factorization of the leading k columns of the matrix A :

```
LAPACKE_?ungqr(matrix_layout, m, m, k, a, lda, tau)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by the leading k columns of the matrix A):

```
LAPACKE_?ungqr(matrix_layout, m, k, k, a, lda, tau)
```

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The order of the unitary matrix Q ($m \geq 0$).
<i>n</i>	The number of columns of Q to be computed ($0 \leq n \leq m$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).
<i>a, tau</i>	Arrays: <i>a</i> and <i>tau</i> are the arrays returned by <i>cgeqrf/zgeqrf</i> or <i>cgeqpf/zgeqpf</i> . The size of <i>a</i> is $\max(1, lda * n)$ for column major layout and $\max(1, lda * m)$ for row major layout . The size of <i>tau</i> must be at least $\max(1, k)$.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

a

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed Q differs from an exactly unitary matrix by a matrix E such that $\|E\|_2 = O(\varepsilon) * \|A\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $16 * m * n * k - 8 * (m + n) * k^2 + (16/3) * k^3$.

If $n = k$, the number is approximately $(8/3) * n^2 * (3m - n)$.

The real counterpart of this routine is [orgqr](#).

?unmqr

*Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by *?geqrf*.*

Syntax

```
lapack_int LAPACKE_cunmqr (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);
```

```
lapack_int LAPACKE_zunmqr (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a rectangular complex matrix C by Q or Q^H , where Q is the unitary matrix Q of the QR factorization formed by the routines [?geqrf](#) or [geqpf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q^H C$, $C Q^H$, $C Q$, or $Q C$ (overwriting the result on C).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a</i> , <i>c</i> , <i>tau</i>	Arrays:

a size $\max(1, lda*k)$ for column major layout, $\max(1, lda*m)$ for row major layout when $side = 'L'$, and $\max(1, lda*n)$ for row major layout when $side = 'R'$ and τ are the arrays returned by `cgeqrf` / `zgeqrf` or `cgeqpz` / `zgeqpz`.

The size of τ must be at least $\max(1, k)$.

c (size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout) contains the m -by- n matrix C .

lda

The leading dimension of a . Constraints:

$lda \geq \max(1, m)$ for column major layout and $lda \geq \max(1, k)$ for row major layout if $side = 'L'$;

$lda \geq \max(1, n)$ for column major layout and $lda \geq \max(1, k)$ for row major layout if $side = 'R'$.

ldc

The leading dimension of c . Constraint:

$ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

c

Overwritten by the product $Q^H C$, $Q^H C$, $C^H Q$, or $C^H Q^H$ (as specified by $side$ and $trans$).

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Application Notes

The real counterpart of this routine is [ormqr](#).

?gelqf

Computes the LQ factorization of a general m-by-n matrix.

Syntax

```
lapack_int LAPACKE_sgelqf (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, float* tau);
```

```
lapack_int LAPACKE_dgelqf (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, double* tau);
```

```
lapack_int LAPACKE_cgelqf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);
```

```
lapack_int LAPACKE_zgelqf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- `mk1.h`

Description

The routine forms the LQ factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>m</i>	The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	The number of columns in A ($n \geq 0$).
<i>a</i>	Array <i>a</i> of size $\max(1, lda \cdot n)$ for column major layout and $\max(1, lda \cdot m)$ for row major layout contains the matrix A .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: The elements on and below the diagonal of the array contain the m -by- $\min(m, n)$ lower trapezoidal matrix L (L is lower triangular if $m \leq n$); the elements above the diagonal, with the array <i>tau</i> , represent the orthogonal matrix Q as a product of elementary reflectors.
<i>tau</i>	Array, size at least $\max(1, \min(m, n))$. Contains scalars that define elementary reflectors for the matrix Q (see Orthogonal Factorizations).

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$(4/3)n^3 \quad \text{if } m = n,$$

$(2/3)n^2(3m-n)$ if $m > n$,

$(2/3)m^2(3n-m)$ if $m < n$.

The number of operations for complex flavors is 4 times greater.

To find the minimum-norm solution of an underdetermined least squares problem minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

<code>?gelqf</code> (this routine)	to factorize $A = L^*Q$;
<code>trsm</code> (a BLAS routine)	to solve $L^*Y = B$ for Y ;
<code>ormlq</code>	to compute $X = (Q_1)^T Y$ (for real matrices);
<code>unmlq</code>	to compute $X = (Q_1)^H Y$ (for complex matrices).

(The columns of the computed X are the minimum-norm solution vectors x . Here A is an m -by- n matrix with $m < n$; Q_1 denotes the first m columns of Q).

To compute the elements of Q explicitly, call

<code>orglq</code>	(for real matrices)
<code>unglq</code>	(for complex matrices).

See Also

[mkl_progress](#)

Matrix Storage Schemes

`?orglq`

Generates the real orthogonal matrix Q of the LQ factorization formed by `?gelqf`.

Syntax

```
lapack_int LAPACKE_sorglq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
float* a, lapack_int lda, const float* tau);
```

```
lapack_int LAPACKE_dorglq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
double* a, lapack_int lda, const double* tau);
```

Include Files

- `mkl.h`

Description

The routine generates the whole or part of n -by- n orthogonal matrix Q of the LQ factorization formed by the routines [gelqf](#). Use this routine after a call to `sgelqf/dgelqf`.

Usually Q is determined from the LQ factorization of an p -by- n matrix A with $n \geq p$. To compute the whole matrix Q , use:

```
info = LAPACKE_?orglq(matrix_layout, n, n, p, a, lda, tau)
```

To compute the leading p rows of Q , which form an orthonormal basis in the space spanned by the rows of A , use:

```
info = LAPACKE_?orglq(matrix_layout, p, n, p, a, lda, tau)
```

To compute the matrix Q^k of the LQ factorization of the leading k rows of A , use:

```
info = LAPACKE_?orglq(matrix_layout, n, n, k, a, lda, tau)
```

To compute the leading k rows of Q^k , which form an orthonormal basis in the space spanned by the leading k rows of A , use:

```
info = LAPACKE_?orgqr(matrix_layout, k, n, k, a, lda, tau)
```

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows of Q to be computed ($0 \leq m \leq n$).
<i>n</i>	The order of the orthogonal matrix Q ($n \geq m$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq m$).
<i>a, tau</i>	Arrays: <i>a</i> (size $\max(1, lda * n)$ for column major layout and $\max(1, lda * m)$ for row major layout) and <i>tau</i> are the arrays returned by <i>sgelqf</i> / <i>dgelqf</i> . The size of <i>tau</i> must be at least $\max(1, k)$.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	Overwritten by m leading rows of the n -by- n orthogonal matrix Q .
----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

Application Notes

The computed Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $4 * m * n * k - 2 * (m + n) * k^2 + (4/3) * k^3$.

If $m = k$, the number is approximately $(2/3) * m^2 * (3n - m)$.

The complex counterpart of this routine is [unglq](#).

?ormlq

Multiplies a real matrix by the orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```
lapack_int LAPACKE_sormlq (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const float* a, lapack_int lda, const float* tau, float* c,
lapack_int ldc);
```

```
lapack_int LAPACKE_dormlq (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const double* a, lapack_int lda, const double* tau, double*
c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the LQ factorization formed by the routine [gelqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (overwriting the result on C).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a</i> , <i>c</i> , <i>tau</i>	Arrays: <i>a</i> and <i>tau</i> are arrays returned by <code>?gelqf</code> . The size of <i>a</i> must be: For <i>side</i> = 'L' and column major layout, $\max(1, lda*m)$. For <i>side</i> = 'R' and column major layout, $\max(1, lda*n)$. For row major layout regardless of <i>side</i> , $\max(1, lda*k)$. The dimension of <i>tau</i> must be at least $\max(1, k)$. <i>c</i> (size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout) contains the m -by- n matrix C .

<i>lda</i>	The leading dimension of <i>a</i> . For column major layout, $lda \geq \max(1, k)$. For row major layout, if <i>side</i> = 'L', $lda \geq \max(1, m)$, or, if <i>side</i> = 'R', $lda \geq \max(1, n)$.
<i>ldc</i>	The leading dimension of <i>c</i> ; $ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , $Q^T C$, C^*Q , or C^*Q^T (as specified by <i>side</i> and <i>trans</i>).
----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is [unmlq](#).

?unglq

Generates the complex unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```
lapack_int LAPACKE_cunglq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);
```

```
lapack_int LAPACKE_zunglq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine generates the whole or part of *n*-by-*n* unitary matrix *Q* of the *LQ* factorization formed by the routines [gelqf](#). Use this routine after a call to [cgelqf](#)/[zgelqf](#).

Usually *Q* is determined from the *LQ* factorization of an *p*-by-*n* matrix *A* with $n < p$. To compute the whole matrix *Q*, use:

```
info = LAPACKE_?unglq(matrix_layout, n, n, p, a, lda, tau)
```

To compute the leading *p* rows of *Q*, which form an orthonormal basis in the space spanned by the rows of *A*, use:

```
info = LAPACKE_?unglq(matrix_layout, p, n, p, a, lda, tau)
```

To compute the matrix Q^k of the *LQ* factorization of the leading *k* rows of *A*, use:

```
info = LAPACKE_?unglq(matrix_layout, n, n, k, a, lda, tau)
```


To compute the leading k rows of Q^k , which form an orthonormal basis in the space spanned by the leading k rows of A , use:

```
info = LAPACKE_?ungqr(matrix_layout, k, n, k, a, lda, tau)
```

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows of Q to be computed ($0 \leq m \leq n$).
<i>n</i>	The order of the unitary matrix Q ($n \geq m$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq m$).
<i>a, tau</i>	Arrays: <i>a</i> (size $\max(1, lda * n)$ for column major layout and $\max(1, lda * m)$ for row major layout) and <i>tau</i> are the arrays returned by cgelqf/zgelqf. The dimension of <i>tau</i> must be at least $\max(1, k)$.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	Overwritten by m leading rows of the n -by- n unitary matrix Q .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

Application Notes

The computed Q differs from an exactly unitary matrix by a matrix E such that $\|E\|_2 = O(\varepsilon) * \|A\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $16 * m * n * k - 8 * (m + n) * k^2 + (16/3) * k^3$.

If $m = k$, the number is approximately $(8/3) * m^2 * (3n - m)$.

The real counterpart of this routine is [orglq](#).

?unmlq

Multiplies a complex matrix by the unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```
lapack_int LAPACKE_cunmlq (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);
```

```
lapack_int LAPACKE_zunmlq (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- `mkl.h`

Description

The routine multiplies a real m -by- n matrix C by Q or Q^H , where Q is the unitary matrix Q of the LQ factorization formed by the routine [gelqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q^H C$, $Q^H C$, $C Q$, or $C Q^H$ (overwriting the result on C).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a</i> , <i>c</i> , <i>tau</i>	Arrays: <i>a</i> and <i>tau</i> are arrays returned by <code>?gelqf</code> . The size of <i>a</i> must be: For <i>side</i> = 'L' and column major layout, $\max(1, lda * m)$. For <i>side</i> = 'R' and column major layout, $\max(1, lda * n)$. For row major layout regardless of <i>side</i> , $\max(1, lda * k)$. The size of <i>tau</i> must be at least $\max(1, k)$. <i>c</i> (size $\max(1, ldc * n)$ for column major layout and $\max(1, ldc * m)$ for row major layout) contains the m -by- n matrix C .
<i>lda</i>	The leading dimension of <i>a</i> . For column major layout, $lda \geq \max(1, k)$. For row major layout, if <i>side</i> = 'L', $lda \geq \max(1, m)$, or, if <i>side</i> = 'R', $lda \geq \max(1, n)$.
<i>ldc</i>	The leading dimension of <i>c</i> ; $ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

c Overwritten by the product Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The real counterpart of this routine is [ormlq](#).

?geqlf

Computes the QL factorization of a general m-by-n matrix.

Syntax

```
lapack_int LAPACKE_sgeqlf (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, float* tau);
```

```
lapack_int LAPACKE_dgeqlf (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, double* tau);
```

```
lapack_int LAPACKE_cgeqlf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);
```

```
lapack_int LAPACKE_zgeqlf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine forms the *QL* factorization of a general *m*-by-*n* matrix *A* (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with *Q* in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

m The number of rows in the matrix *A* ($m \geq 0$).

n	The number of columns in A ($n \geq 0$).
a	Array a of size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout contains the matrix A .
lda	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

a	Overwritten on exit by the factorization data as follows: if $m \geq n$, the lower triangle of the subarray $a(m-n+1:m, 1:n)$ contains the n -by- n lower triangular matrix L ; if $m \leq n$, the elements on and below the $(n-m)$ -th superdiagonal contain the m -by- n lower trapezoidal matrix L ; in both cases, the remaining elements, with the array τ , represent the orthogonal/unitary matrix Q as a product of elementary reflectors.
τ	Array, size at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors for the matrix Q (see Orthogonal Factorizations).

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

Application Notes

Related routines include:

orgql	to generate matrix Q (for real matrices);
ungql	to generate matrix Q (for complex matrices);
ormql	to apply matrix Q (for real matrices);
unmq	to apply matrix Q (for complex matrices).

See Also

[mkl_progress](#)

Matrix Storage Schemes

?orgql

Generates the real matrix Q of the QL factorization formed by ?geqlf.

Syntax

```
lapack_int LAPACKE_sorgql (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
float* a, lapack_int lda, const float* tau);
```

```
lapack_int LAPACKE_dorgql (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
double* a, lapack_int lda, const double* tau);
```

Include Files

- `mk1.h`

Description

The routine generates an m -by- n real matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors $H(i)$ of order m : $Q = H(k) * \dots * H(2) * H(1)$ as returned by the routines [geqlf](#). Use this routine after a call to [sgeqlf/dgeqlf](#).

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>m</code>	The number of rows of the matrix Q ($m \geq 0$).
<code>n</code>	The number of columns of the matrix Q ($m \geq n \geq 0$).
<code>k</code>	The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<code>a, tau</code>	Arrays: a (size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout), tau . On entry, the $(n - k + i)$ th column of a must <i>contain the vector which</i> defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by sgeqlf/dgeqlf in the last k columns of its array argument a ; $tau[i - 1]$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by sgeqlf/dgeqlf ; The size of tau must be at least $\max(1, k)$.
<code>lda</code>	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<code>a</code>	Overwritten by the last n columns of the m -by- m orthogonal matrix Q .
----------------	---

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is [ungql](#).

?ungql

Generates the complex matrix Q of the QL factorization formed by ?geqlf.

Syntax

```
lapack_int LAPACKE_cungql (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);
```

```
lapack_int LAPACKE_zungql (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine generates an m -by- n complex matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors $H(i)$ of order m : $Q = H(k) * \dots * H(2) * H(1)$ as returned by the routines [geqlf/geqlf](#). Use this routine after a call to [cgeqlf/zgeqlf](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows of the matrix Q ($m \geq 0$).
<i>n</i>	The number of columns of the matrix Q ($m \geq n \geq 0$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a, tau</i>	Arrays: <i>a</i> (size $\max(1, lda * n)$ for column major layout and $\max(1, lda * m)$ for row major layout), <i>tau</i> . On entry, the $(n - k + i)$ th column of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by cgeqlf/zgeqlf in the last k columns of its array argument <i>a</i> ; <i>tau</i> [$i - 1$] must contain the scalar factor of the elementary reflector $H(i)$, as returned by cgeqlf/zgeqlf ; The size of <i>tau</i> must be at least $\max(1, k)$.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	Overwritten by the last n columns of the m -by- m unitary matrix Q .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

Application Notes

The real counterpart of this routine is [orgql](#).

?ormql

Multiplies a real matrix by the orthogonal matrix Q of the QL factorization formed by `?geqlf`.

Syntax

```
lapack_int LAPACKE_sormql (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const float* a, lapack_int lda, const float* tau, float* c,
lapack_int ldc);
```

```
lapack_int LAPACKE_dormql (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const double* a, lapack_int lda, const double* tau, double*
c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the QL factorization formed by the routine `geqlf`.

Depending on the parameters *side* and *trans*, the routine `ormql` can form one of the matrix products $Q^T C$, $C^T Q$, or $C^T Q^T$ (overwriting the result over C).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a</i> , <i>tau</i> , <i>c</i>	Arrays: <i>a</i> , <i>tau</i> , <i>c</i> . The size of <i>a</i> must be: For column major layout regardless of <i>side</i> , $\max(1, \text{lda} * k)$. For <i>side</i> = 'L' and row major layout, $\max(1, \text{lda} * m)$. For <i>side</i> = 'R' and row major layout, $\max(1, \text{lda} * n)$.

On entry, the i th column of a must contain the vector which defines the elementary reflector H_i , for $i = 1, 2, \dots, k$, as returned by `sgeqlf/dgeqlf` in the last k columns of its array argument a .

$\tau[i - 1]$ must contain the scalar factor of the elementary reflector H_i , as returned by `sgeqlf/dgeqlf`.

The size of τ must be at least $\max(1, k)$.

c (size $\max(1, ldc * n)$ for column major layout and $\max(1, ldc * m)$ for row major layout) contains the m -by- n matrix C .

lda

The leading dimension of a ;

if $side = 'L'$, $lda \geq \max(1, m)$ for column major layout and $\max(1, k)$ for row major layout ;

if $side = 'R'$, $lda \geq \max(1, n)$ for column major layout and $\max(1, k)$ for row major layout.

ldc

The leading dimension of c ; $ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

c

Overwritten by the product Q^*C , $Q^T C$, C^*Q , or $C^T Q^T$ (as specified by $side$ and $trans$).

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is [unmql](#).

?unmql

Multiplies a complex matrix by the unitary matrix Q of the QL factorization formed by ?geqlf.

Syntax

```
lapack_int LAPACKE_cunmql (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);
```

```
lapack_int LAPACKE_zunmql (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- `mkl.h`

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the unitary matrix Q of the QL factorization formed by the routine [geqlf](#).

Depending on the parameters *side* and *trans*, the routine [unmql](#) can form one of the matrix products $Q^H C$, $C^H Q$, or $C^H Q^H$ (overwriting the result over C).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a</i> , <i>tau</i> , <i>c</i>	Arrays: <i>a</i> , <i>tau</i> , <i>c</i> . The size of <i>a</i> must be: For column major layout regardless of <i>side</i> , $\max(1, lda * k)$. For <i>side</i> = 'L' and row major layout, $\max(1, lda * m)$. For <i>side</i> = 'R' and row major layout, $\max(1, lda * n)$. On entry, the i -th column of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by cgeqlf/zgeqlf in the last k columns of its array argument <i>a</i> . <i>tau</i> [$i - 1$] must contain the scalar factor of the elementary reflector $H(i)$, as returned by cgeqlf/zgeqlf . The size of <i>tau</i> must be at least $\max(1, k)$. <i>c</i> (size $\max(1, ldc * n)$ for column major layout and $\max(1, ldc * m)$ for row major layout) contains the m -by- n matrix C .
<i>lda</i>	The leading dimension of <i>a</i> . If <i>side</i> = 'L', $lda \geq \max(1, m)$ for column major layout and $\max(1, k)$ for row major layout. If <i>side</i> = 'R', $lda \geq \max(1, n)$ for column major layout and $\max(1, k)$ for row major layout.

ldc The leading dimension of *c*; $ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

c Overwritten by the product $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The real counterpart of this routine is [ormql](#).

?gerqf

Computes the RQ factorization of a general m-by-n matrix.

Syntax

```
lapack_int LAPACKE_sgerqf (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, float* tau);

lapack_int LAPACKE_dgerqf (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, double* tau);

lapack_int LAPACKE_cgerqf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);

lapack_int LAPACKE_zgerqf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine forms the *RQ* factorization of a general *m*-by-*n* matrix *A*(see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with *Q* in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>m</code>	The number of rows in the matrix A ($m \geq 0$).
<code>n</code>	The number of columns in A ($n \geq 0$).
<code>a</code>	Array a of size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout contains the m -by- n matrix A .
<code>lda</code>	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<code>a</code>	Overwritten on exit by the factorization data as follows: if $m \leq n$, the upper triangle of the subarray $a(1:m, n-m+1:n)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m-n)$ th subdiagonal contain the m -by- n upper trapezoidal matrix R ; in both cases, the remaining elements, with the array τ , represent the orthogonal/unitary matrix Q as a product of $\min(m, n)$ elementary reflectors.
<code>tau</code>	Array, size at least $\max(1, \min(m, n))$. (See Orthogonal Factorizations .) Contains scalar factors of the elementary reflectors for the matrix Q .

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

Related routines include:

<code>orgqr</code>	to generate matrix Q (for real matrices);
<code>ungrq</code>	to generate matrix Q (for complex matrices);
<code>ormqr</code>	to apply matrix Q (for real matrices);
<code>unmrq</code>	to apply matrix Q (for complex matrices).

See Also

[mkl_progress](#)

Matrix Storage Schemes

?orgqr

Generates the real matrix Q of the RQ factorization formed by *?gerqf*.

Syntax

```
lapack_int LAPACKE_sorgqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
float* a, lapack_int lda, const float* tau);
```

```
lapack_int LAPACKE_dorgqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
double* a, lapack_int lda, const double* tau);
```

Include Files

- mkl.h

Description

The routine generates an m -by- n real matrix with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors $H(i)$ of order n : $Q = H(1) * H(2) * \dots * H(k)$ as returned by the routines [gerqf](#). Use this routine after a call to [sgerqf](#)/[dgerqf](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows of the matrix Q ($m \geq 0$).
<i>n</i>	The number of columns of the matrix Q ($n \geq m$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
<i>a</i> , <i>tau</i>	Arrays: <i>a</i> (size $\max(1, lda * n)$ for column major layout and $\max(1, lda * m)$ for row major layout), <i>tau</i> . On entry, the $(m - k + i)$ -th row of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by sgerqf / dgerqf in the last k rows of its array argument <i>a</i> ; <i>tau</i> [$i - 1$] must contain the scalar factor of the elementary reflector $H(i)$, as returned by sgerqf / dgerqf ; The size of <i>tau</i> must be at least $\max(1, k)$.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	Overwritten by the last m rows of the n -by- n orthogonal matrix Q .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is [ungrq](#).

?ungrq

*Generates the complex matrix Q of the RQ factorization formed by *?gerqf*.*

Syntax

```
lapack_int LAPACKE_cungrq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);

lapack_int LAPACKE_zungrq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine generates an m -by- n complex matrix with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors $H(i)$ of order n : $Q = H(1)^H * H(2)^H * \dots * H(k)^H$ as returned by the routines [gerqf](#). Use this routine after a call to [cgerqf/zgerqf](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows of the matrix Q ($m \geq 0$).
<i>n</i>	The number of columns of the matrix Q ($n \geq m$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
<i>a, tau</i>	Arrays: <i>a</i> (size $\max(1, lda * n)$ for column major layout and $\max(1, lda * m)$ for row major layout), <i>tau</i> . On entry, the $(m - k + i)$ th row of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by cgerqf/zgerqf in the last k rows of its array argument <i>a</i> ; <i>tau</i> [$i - 1$] must contain the scalar factor of the elementary reflector $H(i)$, as returned by cgerqf/zgerqf ; The size of <i>tau</i> must be at least $\max(1, k)$.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	Overwritten by the m last rows of the n -by- n unitary matrix Q .
----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The real counterpart of this routine is [orgrq](#).

?ormrq

Multiplies a real matrix by the orthogonal matrix Q of the RQ factorization formed by ?gerqf.

Syntax

```
lapack_int LAPACKE_sormrq (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const float* a, lapack_int lda, const float* tau, float* c,
lapack_int ldc);
```

```
lapack_int LAPACKE_dormrq (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const double* a, lapack_int lda, const double* tau, double*
c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the real orthogonal matrix defined as a product of k elementary reflectors H_i : $Q = H_1 H_2 \dots H_k$ as returned by the *RQ* factorization routine [gerqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , $Q^T C$, C^*Q , or $C^T Q^T$ (overwriting the result over C).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).

k	<p>The number of elementary reflectors whose product defines the matrix Q. Constraints:</p> <p>$0 \leq k \leq m$, if $side = 'L'$;</p> <p>$0 \leq k \leq n$, if $side = 'R'$.</p>
a, τ, c	<p>Arrays: a(size for $side = 'L'$: $\max(1, lda*m)$ for column major layout and $\max(1, lda*k)$ for row major layout; for $side = 'R'$: $\max(1, lda*n)$ for column major layout and $\max(1, lda*k)$ for row major layout), τ, c (size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout).</p> <p>On entry, the ith row of a must contain the vector which defines the elementary reflector H_i, for $i = 1, 2, \dots, k$, as returned by <code>sgerqf/dgerqf</code> in the last k rows of its array argument a.</p> <p>$\tau[i - 1]$ must contain the scalar factor of the elementary reflector H_i, as returned by <code>sgerqf/dgerqf</code>.</p> <p>The size of τ must be at least $\max(1, k)$.</p> <p>c contains the m-by-n matrix C.</p>
lda	The leading dimension of a ; $lda \geq \max(1, k)$ for column major layout. For row major layout, $lda \geq \max(1, m)$ if $side = 'L'$, and $lda \geq \max(1, n)$ if $side = 'R'$.
ldc	The leading dimension of c ; $ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

c	Overwritten by the product Q^*C , $Q^T C$, C^*Q , or C^*Q^T (as specified by $side$ and $trans$).
-----	---

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is [unmrq](#).

?unmrq

Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by ?gerqf.

Syntax

```
lapack_int LAPACK_cunmrq (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);
```

```
lapack_int LAPACK_zunmrq (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- `mk1.h`

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the complex unitary matrix defined as a product of k elementary reflectors $H(i)$ of order n : $Q = H(1)^{H*} H(2)^{H*} \dots H(k)^{H*}$ as returned by the *RQ* factorization routine [gerqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (overwriting the result over C).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <i>side</i> = 'L'; $0 \leq k \leq n$, if <i>side</i> = 'R'.
<i>a</i> , <i>tau</i> , <i>c</i>	Arrays: <i>a</i> (size for <i>side</i> = 'L': $\max(1, lda*m)$ for column major layout and $\max(1, lda*k)$ for row major layout; for <i>side</i> = 'R': $\max(1, lda*n)$ for column major layout and $\max(1, lda*k)$ for row major layout), <i>tau</i> , <i>c</i> (size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout). On entry, the i th row of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>cgerqf/zgerqf</code> in the last k rows of its array argument <i>a</i> . <i>tau</i> [$i - 1$] must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>cgerqf/zgerqf</code> . The size of <i>tau</i> must be at least $\max(1, k)$. <i>c</i> (size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout) contains the m -by- n matrix C .

<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, k)$ for column major layout. For row major layout, $lda \geq \max(1, m)$ if <i>side</i> = 'L', and $lda \geq \max(1, n)$ if <i>side</i> = 'R'.
<i>ldc</i>	The leading dimension of <i>c</i> ; $ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>c</i>	Overwritten by the product $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (as specified by <i>side</i> and <i>trans</i>).
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The real counterpart of this routine is [ormrq](#).

?tzrzf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

```
lapack_int LAPACKE_stzrzf (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, float* tau);
```

```
lapack_int LAPACKE_dtzrzf (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, double* tau);
```

```
lapack_int LAPACKE_ctzrzf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);
```

```
lapack_int LAPACKE_ztzrzf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine reduces the *m*-by-*n* ($m \leq n$) real/complex upper trapezoidal matrix *A* to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix $A = [A1 \ A2] = [A_{1:m, 1:m}, A_{1:m, m+1:n}]$ is factored as

$$A = [R \ 0] * Z,$$

where *Z* is an *n*-by-*n* orthogonal/unitary matrix, *R* is an *m*-by-*m* upper triangular matrix, and *O* is the *m*-by-*(n-m)* zero matrix.

The *?tzrzf* routine replaces the deprecated *?tzrqf* routine.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>m</i>	The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	The number of columns in <i>A</i> ($n \geq m$).
<i>a</i>	Array <i>a</i> is of size $\max(1, lda * n)$ for column major layout and $\max(1, lda * m)$ for row major layout. The leading <i>m</i> -by- <i>n</i> upper trapezoidal part of the array <i>a</i> contains the matrix <i>A</i> to be factorized.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	Overwritten on exit by the factorization data as follows: the leading <i>m</i> -by- <i>m</i> upper triangular part of <i>a</i> contains the upper triangular matrix <i>R</i> , and elements <i>m</i> + 1 to <i>n</i> of the first <i>m</i> rows of <i>a</i> , with the array <i>tau</i> , represent the orthogonal matrix <i>Z</i> as a product of <i>m</i> elementary reflectors.
<i>tau</i>	Array, size at least $\max(1, m)$. Contains scalar factors of the elementary reflectors for the matrix <i>Z</i> .

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, *Z*(*k*), which is used to introduce zeros into the (*m* - *k* + 1)-th row of *A*, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where for real flavors

$$T(k) = I - \tau u(k) u(k)^T, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

and for complex flavors

$$T(k) = I - \tau u(k) u(k)^H, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

τ is a scalar and $z(k)$ is an l -element vector. τ and $z(k)$ are chosen to annihilate the elements of the k -th row of A_2 .

The scalar τ is returned in the k -th element of τ and the vector $u(k)$ in the k -th row of A , such that the elements of $z(k)$ are stored in the last $m - n$ elements of the k -th row of array a .

The elements of R are returned in the upper triangular part of A .

The matrix Z is given by

$$Z = Z(1) * Z(2) * \dots * Z(m).$$

Related routines include:

ormrz	to apply matrix Q (for real matrices)
unmrz	to apply matrix Q (for complex matrices).

?ormrz

Multiplies a real matrix by the orthogonal matrix defined from the factorization formed by ?tzzrf.

Syntax

```
lapack_int LAPACKE_sormrz (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, const float* a, lapack_int lda, const float*
tau, float* c, lapack_int ldc);
```

```
lapack_int LAPACKE_dormrz (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, const double* a, lapack_int lda, const
double* tau, double* c, lapack_int ldc);
```

Include Files

- `mkl.h`

Description

The `?ormrz` routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the real orthogonal matrix defined as a product of k elementary reflectors $H(i)$ of order n : $Q = H(1) * H(2) * \dots * H(k)$ as returned by the factorization routine `tzrzf`.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q * C$, $Q^T * C$, $C * Q$, or $C * Q^T$ (overwriting the result over C).

The matrix Q is of order m if `side = 'L'` and of order n if `side = 'R'`.

The `?ormrz` routine replaces the deprecated `?latzm` routine.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>side</code>	Must be either <code>'L'</code> or <code>'R'</code> . If <code>side = 'L'</code> , Q or Q^T is applied to C from the left. If <code>side = 'R'</code> , Q or Q^T is applied to C from the right.
<code>trans</code>	Must be either <code>'N'</code> or <code>'T'</code> . If <code>trans = 'N'</code> , the routine multiplies C by Q . If <code>trans = 'T'</code> , the routine multiplies C by Q^T .
<code>m</code>	The number of rows in the matrix C ($m \geq 0$).
<code>n</code>	The number of columns in C ($n \geq 0$).
<code>k</code>	The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <code>side = 'L'</code> ; $0 \leq k \leq n$, if <code>side = 'R'</code> .
<code>l</code>	The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints: $0 \leq l \leq m$, if <code>side = 'L'</code> ; $0 \leq l \leq n$, if <code>side = 'R'</code> .
<code>a, tau, c</code>	Arrays: a (size for <code>side = 'L'</code> : $\max(1, lda * m)$ for column major layout and $\max(1, lda * k)$ for row major layout; for <code>side = 'R'</code> : $\max(1, lda * b)$ for column major layout and $\max(1, lda * k)$ for row major layout), tau , c (size $\max(1, ldc * n)$ for column major layout and $\max(1, ldc * m)$ for row major layout). On entry, the i th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>stzrzf/dtzrzf</code> in the last k rows of its array argument a . $tau[i - 1]$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>stzrzf/dtzrzf</code> . The size of tau must be at least $\max(1, k)$.

c contains the m -by- n matrix C .

lda The leading dimension of a ; $lda \geq \max(1, k)$ for column major layout. For row major layout, $lda \geq \max(1, m)$ if $side = 'L'$, and $lda \geq \max(1, n)$ if $side = 'R'$.

ldc The leading dimension of c ; $ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

c Overwritten by the product Q^*C , $Q^T C$, C^*Q , or C^*Q^T (as specified by $side$ and $trans$).

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is [unmrz](#).

?unmrz

Multiplies a complex matrix by the unitary matrix defined from the factorization formed by ?tzzrf.

Syntax

```
lapack_int LAPACKE_cunmrz (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, const lapack_complex_float* a, lapack_int
lda, const lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);
```

```
lapack_int LAPACKE_zunmrz (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, const lapack_complex_double* a, lapack_int
lda, const lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the unitary matrix defined as a product of k elementary reflectors $H(i)$:

$Q = H(1)^H \cdot H(2)^H \cdot \dots \cdot H(k)^H$ as returned by the factorization routine [tzzrf](#).

Depending on the parameters $side$ and $trans$, the routine can form one of the matrix products Q^*C , $Q^H C$, C^*Q , or C^*Q^H (overwriting the result over C).

The matrix Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>side</i>	<p>Must be either 'L' or 'R'.</p> <p>If <i>side</i> = 'L', Q or Q^H is applied to C from the left.</p> <p>If <i>side</i> = 'R', Q or Q^H is applied to C from the right.</p>
<i>trans</i>	<p>Must be either 'N' or 'C'.</p> <p>If <i>trans</i> = 'N', the routine multiplies C by Q.</p> <p>If <i>trans</i> = 'C', the routine multiplies C by Q^H.</p>
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>k</i>	<p>The number of elementary reflectors whose product defines the matrix Q. Constraints:</p> <p>$0 \leq k \leq m$, if <i>side</i> = 'L';</p> <p>$0 \leq k \leq n$, if <i>side</i> = 'R'.</p>
<i>l</i>	<p>The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints:</p> <p>$0 \leq l \leq m$, if <i>side</i> = 'L';</p> <p>$0 \leq l \leq n$, if <i>side</i> = 'R'.</p>
<i>a, tau, c</i>	<p>Arrays: <i>a</i> (size for <i>side</i> = 'L': $\max(1, lda*m)$ for column major layout and $\max(1, lda*k)$ for row major layout; for <i>side</i> = 'R': $\max(1, lda*b)$ for column major layout and $\max(1, lda*k)$ for row major layout), <i>tau</i>, <i>c</i> (size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout).</p> <p>On entry, the <i>i</i>th row of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>ctzrzf/ztzrzf</code> in the last k rows of its array argument <i>a</i>.</p> <p><i>tau</i>[<i>i</i> - 1] must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>ctzrzf/ztzrzf</code>.</p> <p>The size of <i>tau</i> must be at least $\max(1, k)$.</p> <p><i>c</i> contains the m-by-n matrix C.</p>
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, k)$ for column major layout. For row major layout, $lda \geq \max(1, m)$ if <i>side</i> = 'L', and $lda \geq \max(1, n)$ if <i>side</i> = 'R'.
<i>ldc</i>	The leading dimension of <i>c</i> ; $ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

c Overwritten by the product Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The real counterpart of this routine is [ormrz](#).

?ggqrf

Computes the generalized QR factorization of two matrices.

Syntax

```
lapack_int LAPACKE_sggqrf (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
float* a, lapack_int lda, float* taua, float* b, lapack_int ldb, float* taub);
```

```
lapack_int LAPACKE_dggqrf (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
double* a, lapack_int lda, double* taua, double* b, lapack_int ldb, double* taub);
```

```
lapack_int LAPACKE_cggqrf (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* taua,
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* taub);
```

```
lapack_int LAPACKE_zggqrf (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* taua,
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* taub);
```

Include Files

- mkl.h

Description

The routine forms the generalized QR factorization of an n -by- m matrix A and an n -by- p matrix B as $A = Q^*R$, $B = Q^*T^*Z$, where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{matrix} & m \\ & \\ n - m & \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} \end{matrix}, \quad \text{if } n \geq m$$

or

$$R = \begin{matrix} n & m - n \\ \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} \end{matrix}, \quad \text{if } n < m$$

where R_{11} is upper triangular, and

$$T = \begin{matrix} p - n & n \\ \begin{pmatrix} 0 & T_{12} \end{pmatrix} \end{matrix}, \quad \text{if } n \leq p,$$

$$T = \begin{matrix} & p \\ n - p & \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \\ p \end{matrix}, \quad \text{if } n > p,$$

where T_{12} or T_{21} is a p -by- p upper triangular matrix.

In particular, if B is square and nonsingular, the GQR factorization of A and B implicitly gives the QR factorization of $B^{-1}A$ as:

$$B^{-1}A = Z^T (T^{-1}R) \quad (\text{for real flavors}) \quad \text{or} \quad B^{-1}A = Z^{H*} (T^{-1}R) \quad (\text{for complex flavors}).$$

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>n</i>	The number of rows of the matrices A and B ($n \geq 0$).
<i>m</i>	The number of columns in A ($m \geq 0$).
<i>p</i>	The number of columns in B ($p \geq 0$).
<i>a</i> , <i>b</i>	Array a of size $\max(1, lda*m)$ for column major layout and $\max(1, lda*n)$ for row major layout contains the matrix A . Array b of size $\max(1, ldb*p)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the matrix B .
<i>lda</i>	The leading dimension of a ; at least $\max(1, n)$ for column major layout and at least $\max(1, m)$ for row major layout.
<i>ldb</i>	The leading dimension of b ; at least $\max(1, n)$ for column major layout and at least $\max(1, p)$ for row major layout.

Output Parameters

a, b	<p>Overwritten by the factorization data as follows:</p> <p>on exit, the elements on and above the diagonal of the array a contain the $\min(n,m)$-by-m upper trapezoidal matrix R (R is upper triangular if $n \geq m$); the elements below the diagonal, with the array τ_{aua}, represent the orthogonal/unitary matrix Q as a product of $\min(n,m)$ elementary reflectors ;</p> <p>if $n \leq p$, the upper triangle of the subarray $b(1:n, p-n+1:p)$ contains the n-by-n upper triangular matrix T;</p> <p>if $n > p$, the elements on and above the $(n-p)$th subdiagonal contain the n-by-p upper trapezoidal matrix T; the remaining elements, with the array τ_{aub}, represent the orthogonal/unitary matrix Z as a product of elementary reflectors.</p>
τ_{aua}, τ_{aub}	<p>Arrays, size at least $\max(1, \min(n, m))$ for τ_{aua} and at least $\max(1, \min(n, p))$ for τ_{aub}. The array τ_{aua} contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q.</p> <p>The array τ_{aub} contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(1)H(2) \dots H(k)$, where $k = \min(n, m)$.

Each $H(i)$ has the form

$H(i) = I - \tau_a * v * v^T$ for real flavors, or

$H(i) = I - \tau_a * v * v^H$ for complex flavors,

where τ_a is a real/complex scalar, and v is a real/complex vector with $v_j = 0$ for $1 \leq j \leq i - 1$, $v_i = 1$.

On exit, for $i + 1 \leq j \leq n$, v_j is stored in $a[(j - 1) + (i - 1) * lda]$ for column major layout and in $a[(j - 1) * lda + (i - 1)]$ for row major layout and τ_a is stored in $\tau_{aua}[i - 1]$

The matrix Z is represented as a product of elementary reflectors

$Z = H(1)H(2) \dots H(k)$, where $k = \min(n, p)$.

Each $H(i)$ has the form

$H(i) = I - \tau_b * v * v^T$ for real flavors, or

$H(i) = I - \tau_b * v * v^H$ for complex flavors,

where τ_b is a real/complex scalar, and v is a real/complex vector with $v_{p-k+1} = 1$, $v_j = 0$ for $p - k + 1 \leq j \leq p - 1$.

On exit, for $1 \leq j \leq p - k + i - 1$, v_j is stored in $b[(n - k + i - 1) + (j - 1) * ldb]$ for column major layout and in $b[(n - k + i - 1) * ldb + (j - 1)]$ for row major layout and τ_b is stored in $\tau_{aub}[i - 1]$.

?ggrqf

Computes the generalized RQ factorization of two matrices.

Syntax

```
lapack_int LAPACKESggrqf (int matrix_layout, lapack_int m, lapack_int p, lapack_int n,
float* a, lapack_int lda, float* taua, float* b, lapack_int ldb, float* taub);
```

```
lapack_int LAPACKEdggrqf (int matrix_layout, lapack_int m, lapack_int p, lapack_int n,
double* a, lapack_int lda, double* taua, double* b, lapack_int ldb, double* taub);
```

```
lapack_int LAPACKEcggrqf (int matrix_layout, lapack_int m, lapack_int p, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* taua,
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* taub);
```

```
lapack_int LAPACKZggrqf (int matrix_layout, lapack_int m, lapack_int p, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* taua,
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* taub);
```

Include Files

- mkl.h

Description

The routine forms the generalized RQ factorization of an m -by- n matrix A and an p -by- n matrix B as $A = R^*Q$, $B = Z^*T^*Q$, where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{matrix} n-m & m \\ m & \begin{pmatrix} 0 & R_{12} \end{pmatrix} \end{matrix}, \quad \text{if } m \leq n,$$

or

$$R = \begin{matrix} & n \\ m-n & \begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix} \\ n & \end{matrix}, \quad \text{if } m > n,$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{matrix} & n \\ & n \\ p - n & \begin{pmatrix} T_{11} \\ 0 \end{pmatrix} \end{matrix}, \quad \text{if } p \geq n,$$

or

$$T = \begin{matrix} & p & n - p \\ p & \begin{pmatrix} T_{11} & T_{12} \end{pmatrix} \end{matrix}, \quad \text{if } p < n,$$

where T_{11} is upper triangular.

In particular, if B is square and nonsingular, the *GRQ* factorization of A and B implicitly gives the *RQ* factorization of $A*B^{-1}$ as:

$$A*B^{-1} = (R*T^{-1}) * Z^T \text{ (for real flavors) or } A*B^{-1} = (R*T^{-1}) * Z^H \text{ (for complex flavors).}$$

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>m</i>	The number of rows of the matrix A ($m \geq 0$).
<i>p</i>	The number of rows in B ($p \geq 0$).
<i>n</i>	The number of columns of the matrices A and B ($n \geq 0$).
<i>a, b</i>	Arrays: <i>a</i> (size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout) contains the m -by- n matrix A . <i>b</i> (size $\max(1, ldb*n)$ for column major layout and $\max(1, ldb*p)$ for row major layout) contains the p -by- n matrix B .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
<i>ldb</i>	The leading dimension of <i>b</i> ; at least $\max(1, p)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a, b</i>	Overwritten by the factorization data as follows: on exit, if $m \leq n$, element R_{ij} ($1 \leq i \leq j \leq m$) of upper triangular matrix R is stored in $a[(i - 1) + (n - m + j - 1)*lda]$ for column major layout and in $a[(i - 1)*lda + (n - m + j - 1)]$ for row major layout.
-------------	--

if $m > n$, the elements on and above the $(m-n)$ th subdiagonal contain the m -by- n upper trapezoidal matrix R ;

the remaining elements, with the array *taua*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

The elements on and above the diagonal of the array *b* contain the $\min(p,n)$ -by- n upper trapezoidal matrix T (T is upper triangular if $p \geq n$); the elements below the diagonal, with the array *taub*, represent the orthogonal/unitary matrix Z as a product of elementary reflectors.

taua, *taub*

Arrays, size at least $\max(1, \min(m, n))$ for *taua* and at least $\max(1, \min(p, n))$ for *taub*.

The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q .

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z .

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(1)H(2) \dots H(k)$, where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - \tau a u a * v * v^T$ for real flavors, or

$H(i) = I - \tau a u a * v * v^H$ for complex flavors,

where *taua* is a real/complex scalar, and *v* is a real/complex vector with $v_{n-k+i} = 1$, $v_{n-k+i+1:n} = 0$.

On exit, $v_{1:n-k+i-1}$ is stored in $a(m-k+i, 1:n-k+i-1)$ and *taua* is stored in *taua*[$i - 1$].

The matrix Z is represented as a product of elementary reflectors

$Z = H(1)H(2) \dots H(k)$, where $k = \min(p, n)$.

Each $H(i)$ has the form

$H(i) = I - \tau a u b * v * v^T$ for real flavors, or

$H(i) = I - \tau a u b * v * v^H$ for complex flavors,

where *taub* is a real/complex scalar, and *v* is a real/complex vector with $v_{1:i-1} = 0$, $v_i = 1$.

On exit, $v_{i+1:p}$ is stored in $b(i+1:p, i)$ and *taub* is stored in *taub*[$i - 1$].

?tpqrt

Computes a blocked QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for Q .

Syntax

```
lapack_int LAPACKE_stpqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int l,
lapack_int nb, float* a, lapack_int lda, float* b, lapack_int ldb, float* t, lapack_int
ldt);
```

```
lapack_int LAPACKE_dtpqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int l,
lapack_int nb, double* a, lapack_int lda, double* b, lapack_int ldb, double* t,
lapack_int ldt);
```

```
lapack_int LAPACKE_ctpqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int l,
lapack_int nb, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b,
lapack_int ldb, lapack_complex_float* t, lapack_int ldt);
```

```
lapack_int LAPACKE_ztpqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int l,
lapack_int nb, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b,
lapack_int ldb, lapack_complex_double* t, lapack_int ldt);
```

Include Files

- mkl.h

Description

The input matrix C is an $(n+m)$ -by- n matrix

$$C = \begin{bmatrix} A \\ B \end{bmatrix} \leftarrow \begin{matrix} n \times n \text{ upper triangular} \\ m \times n \text{ pentagonal} \end{matrix}$$

where A is an n -by- n upper triangular matrix, and B is an m -by- n pentagonal matrix consisting of an $(m-1)$ -by- n rectangular matrix $B1$ on top of an 1 -by- n upper trapezoidal matrix $B2$:

$$B = \begin{bmatrix} B1 \\ B2 \end{bmatrix} \leftarrow \begin{matrix} (m-1) \times n \text{ rectangular} \\ 1 \times n \text{ upper trapezoidal} \end{matrix}$$

The upper trapezoidal matrix $B2$ consists of the first l rows of an n -by- n upper triangular matrix, where $0 \leq l \leq \min(m, n)$. If $l=0$, B is an m -by- n rectangular matrix. If $m=l=n$, B is upper triangular. The elementary reflectors $H(i)$ are stored in the i th column below the diagonal in the $(n+m)$ -by- n input matrix C . The structure of vectors defining the elementary reflectors is illustrated by:

$$\begin{bmatrix} I \\ V \end{bmatrix} \leftarrow \begin{matrix} n \times n \text{ identity} \\ m \times n \text{ pentagonal} \end{matrix}$$

The elements of the unit matrix I are not stored. Thus, V contains all of the necessary information, and is returned in array b .

NOTE

Note that V has the same form as B :

$$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix} \leftarrow \begin{matrix} (m-l) \times n \text{ rectangular} \\ l \times n \text{ upper trapezoidal} \end{matrix}$$

The columns of V represent the vectors which define the $H(i)$ s.

The number of blocks is $k = \text{ceiling}(n/nb)$, where each block is of order nb except for the last block, which is of order $ib = n - (k-1)*nb$. For each of the k blocks, an upper triangular block reflector factor is computed: $T1, T2, \dots, Tk$. The nb -by- nb (ib -by- ib for the last block) T s are stored in the nb -by- n array t as

$$t = [T1T2 \dots Tk] .$$

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>m</code>	The total number of rows in the matrix B ($m \geq 0$).
<code>n</code>	The number of columns in B and the order of the triangular matrix A ($n \geq 0$).
<code>l</code>	The number of rows of the upper trapezoidal part of B ($\min(m, n) \geq l \geq 0$).
<code>nb</code>	The block size to use in the blocked QR factorization ($n \geq nb \geq 1$).
<code>a, b</code>	Arrays: a size $lda*n$ contains the n -by- n upper triangular matrix A . b size $\max(1, ldb*n)$ for column major layout and $\max(1, ldb*m)$ for row major layout, the pentagonal m -by- n matrix B . The first $(m-l)$ rows contain the rectangular $B1$ matrix, and the next l rows contain the upper trapezoidal $B2$ matrix.
<code>lda</code>	The leading dimension of a ; at least $\max(1, n)$.

<i>ldb</i>	The leading dimension of <i>b</i> ; at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.
<i>ldt</i>	The leading dimension of <i>t</i> ; at least <i>nb</i> for column major layout and at least $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	The elements on and above the diagonal of the array contain the upper triangular matrix <i>R</i> .
<i>b</i>	The pentagonal matrix <i>V</i> .
<i>t</i>	Array, size <i>ldt</i> * <i>n</i> for column major layout and <i>ldt</i> * <i>nb</i> for row major layout. The upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?tpmqrt

Applies a real or complex orthogonal matrix obtained from a "triangular-pentagonal" complex block reflector to a general real or complex matrix, which consists of two blocks.

Syntax

```
lapack_int LAPACKE_stpmqrt (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, lapack_int nb, const float* v, lapack_int ldv,
const float* t, lapack_int ldt, float* a, lapack_int lda, float* b, lapack_int ldb);

lapack_int LAPACKE_dtpmqrt (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, lapack_int nb, const double* v, lapack_int
ldv, const double* t, lapack_int ldt, double* a, lapack_int lda, double* b, lapack_int
ldb);

lapack_int LAPACKE_ctpmqrt (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, lapack_int nb, const lapack_complex_float* v,
lapack_int ldv, const lapack_complex_float* t, lapack_int ldt, lapack_complex_float* a,
lapack_int lda, lapack_complex_float* b, lapack_int ldb);

lapack_int LAPACKE_ztpmqrt (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, lapack_int nb, const lapack_complex_double*
v, lapack_int ldv, const lapack_complex_double* t, lapack_int ldt,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb);
```

Include Files

- mkl.h

Description

The columns of the pentagonal matrix V contain the elementary reflectors $H(1), H(2), \dots, H(k)$; V is composed of a rectangular block $V1$ and a trapezoidal block $V2$:

$$V = \begin{bmatrix} V1 & I \\ V2 & \end{bmatrix}$$

The size of the trapezoidal block $V2$ is determined by the parameter l , where $0 \leq l \leq k$. $V2$ is upper trapezoidal, consisting of the first l rows of a k -by- k upper triangular matrix.

If $l=k$, $V2$ is upper triangular;

If $l=0$, there is no trapezoidal block, so $V = V1$ is rectangular.

If $side = 'L'$:

$$C = \begin{bmatrix} A \\ B \end{bmatrix}$$

where A is k -by- n , B is m -by- n and V is m -by- k .

If $side = 'R'$:

$$C = \begin{bmatrix} A & B \end{bmatrix}$$

where A is m -by- k , B is m -by- n and V is n -by- k .

The real/complex orthogonal matrix Q is formed from V and T .

If $trans='N'$ and $side='L'$, c contains $Q * C$ on exit.

If $trans='T'$ and $side='L'$, c contains $Q^T * C$ on exit.

If $trans='C'$ and $side='L'$, c contains $Q^H * C$ on exit.

If $trans='N'$ and $side='R'$, c contains $C * Q$ on exit.

If $trans='T'$ and $side='R'$, c contains $C * Q^T$ on exit.

If $trans='C'$ and $side='R'$, C contains $C * Q^H$ on exit.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	<p>$= 'L'$: apply Q, Q^T, or Q^H from the left.</p> <p>$= 'R'$: apply Q, Q^T, or Q^H from the right.</p>
<i>trans</i>	<p>$= 'N'$, no transpose, apply Q.</p> <p>$= 'T'$, transpose, apply Q^T.</p> <p>$= 'C'$, transpose, apply Q^H.</p>
<i>m</i>	The number of rows in the matrix B , ($m \geq 0$).
<i>n</i>	The number of columns in the matrix B , ($n \geq 0$).
<i>k</i>	The number of elementary reflectors whose product defines the matrix Q , ($k \geq 0$).
<i>l</i>	The order of the trapezoidal part of V ($k \geq l \geq 0$).
<i>nb</i>	The block size used for the storage of t , $k \geq nb \geq 1$. This must be the same value of nb used to generate t in tpqrt .
<i>v</i>	<p>Size $ldv*k$ for column major layout; $ldv*m$ for row major layout and $side = 'L'$, $ldv*n$ for row major layout and $side = 'R'$.</p> <p>The ith column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by tpqrt in array argument b.</p>
<i>ldv</i>	<p>The leading dimension of the array v.</p> <p>If $side = 'L'$, ldv must be at least $\max(1, m)$ for column major layout and $\max(1, k)$ for row major layout;</p> <p>If $side = 'R'$, ldv must be at least $\max(1, n)$ for column major layout and $\max(1, k)$ for row major layout.</p>
<i>t</i>	<p>Array, size $ldt*k$ for column major layout and $ldt*nb$ for row major layout.</p> <p>The upper triangular factors of the block reflectors as returned by tpqrt</p>
<i>ldt</i>	The leading dimension of the array t . ldt must be at least nb for column major layout and $\max(1, k)$ for row major layout.
<i>a</i>	<p>If $side = 'L'$, size $lda*n$ for column major layout and $lda*k$ for row major layout ..</p> <p>If $side = 'R'$, size $lda*k$ for column major layout and $lda*m$ for row major layout ..</p> <p>The k-by-n or m-by-k matrix A.</p>
<i>lda</i>	The leading dimension of the array a .

If $side = 'L'$, lda must be at least $\max(1, k)$ for column major layout and $\max(1, n)$ for row major layout.

If $side = 'R'$, lda must be at least $\max(1, m)$ for column major layout and $\max(1, k)$ for row major layout.

b Size $ldb*n$ for column major layout and $ldb*m$ for row major layout.

The m -by- n matrix B .

ldb The leading dimension of the array b . ldb must be at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

a Overwritten by the corresponding block of the product $Q*C$, $C*Q$, Q^T*C , $C*Q^T$, Q^H*C , or $C*Q^H$.

b Overwritten by the corresponding block of the product $Q*C$, $C*Q$, Q^T*C , $C*Q^T$, Q^H*C , or $C*Q^H$.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Singular Value Decomposition: LAPACK Computational Routines

This topic describes LAPACK routines for computing the *singular value decomposition* (SVD) of a general m -by- n matrix A :

$$A = U\Sigma V^H.$$

In this decomposition, U and V are unitary (for complex A) or orthogonal (for real A); Σ is an m -by- n diagonal matrix with real diagonal elements σ_i :

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m, n)} \geq 0.$$

The diagonal elements σ_i are *singular values* of A . The first $\min(m, n)$ columns of the matrices U and V are, respectively, *left* and *right singular vectors* of A . The singular values and singular vectors satisfy

$$AV_i = \sigma_i U_i \text{ and } A^H U_i = \sigma_i V_i$$

where U_i and V_i are the i -th columns of U and V , respectively.

To find the SVD of a general matrix A , call the LAPACK routine `?gebrd` or `?gbbbrd` for reducing A to a bidiagonal matrix B by a unitary (orthogonal) transformation: $A = QBP^H$. Then call `?bdsqr`, which forms the SVD of a bidiagonal matrix: $B = U_1 \Sigma V_1^H$.

Thus, the sought-for SVD of A is given by $A = U\Sigma V^H = (QU_1)\Sigma(V_1^H P^H)$.

Table "Computational Routines for Singular Value Decomposition (SVD)" lists LAPACK routines that perform singular value decomposition of matrices.

Computational Routines for Singular Value Decomposition (SVD)

Operation	Real matrices	Complex matrices
Reduce A to a bidiagonal matrix B : $A = QBP^H$ (full storage)	<code>?gebrd</code>	<code>?gebrd</code>

Operation	Real matrices	Complex matrices
Reduce A to a bidiagonal matrix B : $A = QBP^H$ (band storage)	?gbbrd	?gbbrd
Generate the orthogonal (unitary) matrix Q or P	?orgbr	?ungbr
Apply the orthogonal (unitary) matrix Q or P	?ormbr	?unmbr
Form singular value decomposition of the bidiagonal matrix B : $B = U\Sigma V^H$?bdsqr ?bdsdc	?bdsqr

Decision Tree: Singular Value Decomposition

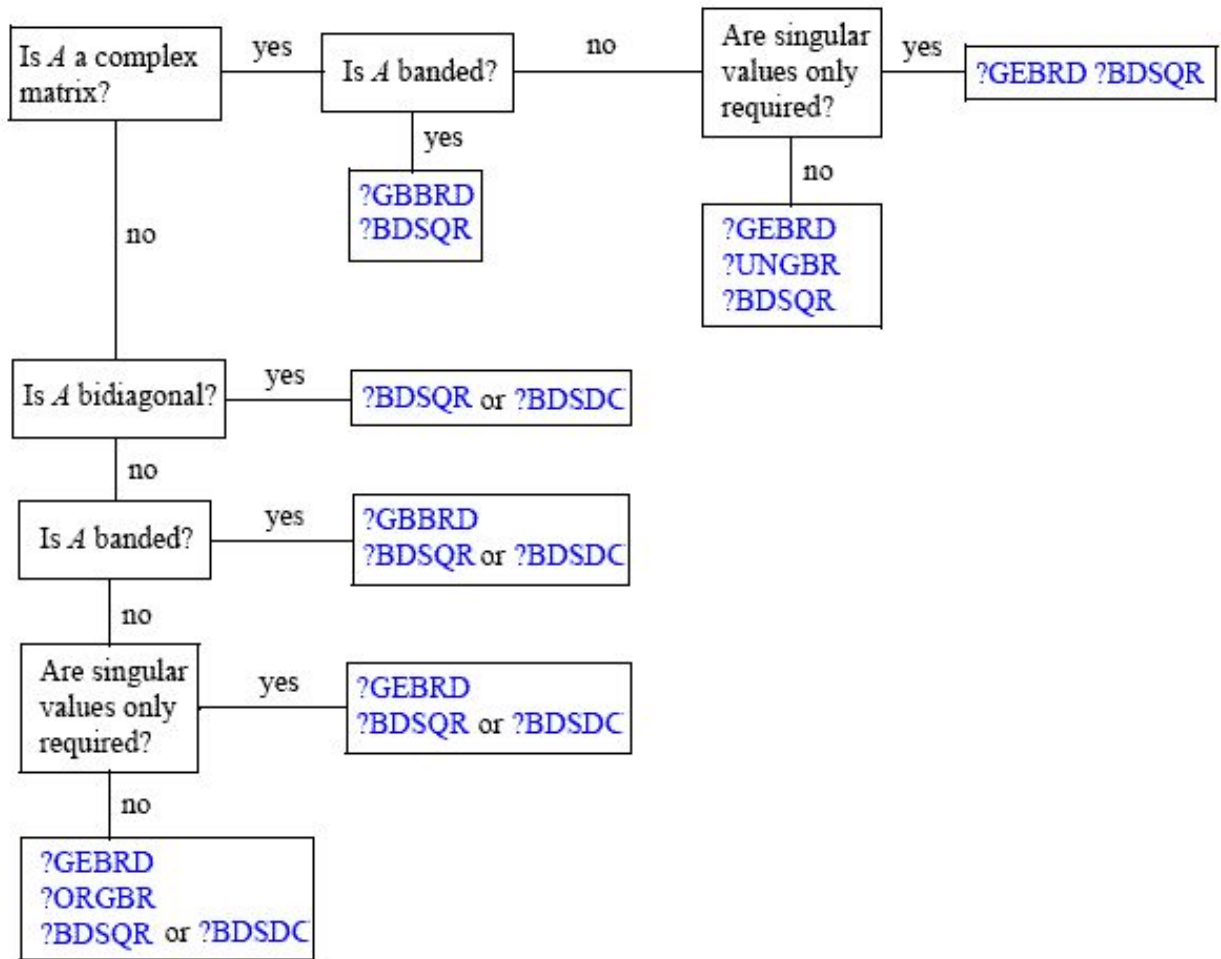


Figure "Decision Tree: Singular Value Decomposition" presents a decision tree that helps you choose the right sequence of routines for SVD, depending on whether you need singular values only or singular vectors as well, whether A is real or complex, and so on.

You can use the SVD to find a minimum-norm solution to a (possibly) rank-deficient least squares problem of minimizing $\|Ax - b\|^2$. The effective rank k of the matrix A can be determined as the number of singular values which exceed a suitable threshold. The minimum-norm solution is

$$x = V_k(\Sigma_k)^{-1}c$$

where Σ_k is the leading k -by- k submatrix of Σ , the matrix V_k consists of the first k columns of $V = PV_1$, and the vector c consists of the first k elements of $U^H b = U_1^H Q^H b$.

?gebrd

Reduces a general matrix to bidiagonal form.

Syntax

```
lapack_int LAPACKE_sgebrd( int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, float* d, float* e, float* tauq, float* tauq );

lapack_int LAPACKE_dgebrd( int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, double* d, double* e, double* tauq, double* tauq );

lapack_int LAPACKE_cgebrd( int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, float* d, float* e, lapack_complex_float*
tauq, lapack_complex_float* tauq );

lapack_int LAPACKE_zgebrd( int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, double* d, double* e, lapack_complex_double*
tauq, lapack_complex_double* tauq );
```

Include Files

- mkl.h

Description

The routine reduces a general m -by- n matrix A to a bidiagonal matrix B by an orthogonal (unitary) transformation.

If $m \geq n$, the reduction is given by $A = QBP^H = \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P_H$,

where B_1 is an n -by- n upper diagonal matrix, Q and P are orthogonal or, for a complex A , unitary matrices; Q_1 consists of the first n columns of Q .

If $m < n$, the reduction is given by

$$A = Q^* B^* P^H = Q^* (B_1 \ 0) P^H = Q_1^* B_1^* P_1^H,$$

where B_1 is an m -by- m lower diagonal matrix, Q and P are orthogonal or, for a complex A , unitary matrices; P_1 consists of the first m columns of P .

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices Q and P in this representation:

If the matrix A is real,

- to compute Q and P explicitly, call [orgbr](#).
- to multiply a general matrix by Q or P , call [ormbr](#).

If the matrix A is complex,

- to compute Q and P explicitly, call [ungbr](#).
- to multiply a general matrix by Q or P , call [unmbr](#).

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>m</code>	The number of rows in the matrix A ($m \geq 0$).
<code>n</code>	The number of columns in A ($n \geq 0$).

<i>a</i>	Arrays: <i>a</i> (size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout) contains the matrix <i>A</i> .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	<p>If $m \geq n$, the diagonal and first super-diagonal of <i>a</i> are overwritten by the upper bidiagonal matrix <i>B</i>. The elements below the diagonal, with the array <i>tauq</i>, represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors, and the elements above the first superdiagonal, with the array <i>taup</i>, represent the orthogonal matrix <i>P</i> as a product of elementary reflectors.</p> <p>If $m < n$, the diagonal and first sub-diagonal of <i>a</i> are overwritten by the lower bidiagonal matrix <i>B</i>. The elements below the first subdiagonal, with the array <i>tauq</i>, represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors, and the elements above the diagonal, with the array <i>taup</i>, represent the orthogonal matrix <i>P</i> as a product of elementary reflectors.</p>
<i>d</i>	<p>Array, size at least $\max(1, \min(m, n))$.</p> <p>Contains the diagonal elements of <i>B</i>.</p>
<i>e</i>	<p>Array, size at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of <i>B</i>.</p>
<i>tauq</i> , <i>taup</i>	<p>Arrays, size at least $\max(1, \min(m, n))$. The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrices <i>P</i> and <i>Q</i>.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrices *Q*, *B*, and *P* satisfy $QBPH = A + E$, where $\|E\|_2 = c(n)\varepsilon \|A\|_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

The approximate number of floating-point operations for real flavors is

$$(4/3)*n^2*(3*m - n) \text{ for } m \geq n,$$

$$(4/3)*m^2*(3*n - m) \text{ for } m < n.$$

The number of operations for complex flavors is four times greater.

If *n* is much less than *m*, it can be more efficient to first form the *QR* factorization of *A* by calling [geqrf](#) and then reduce the factor *R* to bidiagonal form. This requires approximately $2*n^2*(m + n)$ floating-point operations.

If m is much less than n , it can be more efficient to first form the LQ factorization of A by calling [gelqf](#) and then reduce the factor L to bidiagonal form. This requires approximately $2*m^2*(m + n)$ floating-point operations.

?gbbbrd

Reduces a general band matrix to bidiagonal form.

Syntax

```
lapack_int LAPACKE_sgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int ncc, lapack_int kl, lapack_int ku, float* ab, lapack_int ldab, float* d,
float* e, float* q, lapack_int ldq, float* pt, lapack_int ldpt, float* c, lapack_int
ldc );

lapack_int LAPACKE_dgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int ncc, lapack_int kl, lapack_int ku, double* ab, lapack_int ldab, double* d,
double* e, double* q, lapack_int ldq, double* pt, lapack_int ldpt, double* c, lapack_int
ldc );

lapack_int LAPACKE_cgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int ncc, lapack_int kl, lapack_int ku, lapack_complex_float* ab, lapack_int
ldab, float* d, float* e, lapack_complex_float* q, lapack_int ldq,
lapack_complex_float* pt, lapack_int ldpt, lapack_complex_float* c, lapack_int ldc );

lapack_int LAPACKE_zgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int ncc, lapack_int kl, lapack_int ku, lapack_complex_double* ab, lapack_int
ldab, double* d, double* e, lapack_complex_double* q, lapack_int ldq,
lapack_complex_double* pt, lapack_int ldpt, lapack_complex_double* c, lapack_int ldc );
```

Include Files

- mkl.h

Description

The routine reduces an m -by- n band matrix A to upper bidiagonal matrix B : $A = Q*B*P^H$. Here the matrices Q and P are orthogonal (for real A) or unitary (for complex A). They are determined as products of Givens rotation matrices, and may be formed explicitly by the routine if required. The routine can also update a matrix C as follows: $C = Q^H*C$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>vect</i>	Must be 'N' or 'Q' or 'P' or 'B'. If <i>vect</i> = 'N', neither Q nor P^H is generated. If <i>vect</i> = 'Q', the routine generates the matrix Q . If <i>vect</i> = 'P', the routine generates the matrix P^H . If <i>vect</i> = 'B', the routine generates both Q and P^H .
<i>m</i>	The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	The number of columns in A ($n \geq 0$).

<i>ncc</i>	The number of columns in <i>C</i> ($ncc \geq 0$).
<i>kl</i>	The number of sub-diagonals within the band of <i>A</i> ($kl \geq 0$).
<i>ku</i>	The number of super-diagonals within the band of <i>A</i> ($ku \geq 0$).
<i>ab, c</i>	<p>Arrays:</p> <p><i>ab</i>(size $\max(1, ldab*n)$ for column major layout and $\max(1, ldab*m)$ for row major layout) contains the matrix <i>A</i> in band storage (see Matrix Storage Schemes).</p> <p><i>c</i>(size $\max(1, ldc*ncc)$ for column major layout and $\max(1, ldc*m)$ for row major layout) contains an <i>m</i>-by-<i>ncc</i> matrix <i>C</i>.</p> <p>If $ncc = 0$, the array <i>c</i> is not referenced.</p>
<i>ldab</i>	The leading dimension of the array <i>ab</i> ($ldab \geq kl + ku + 1$).
<i>ldq</i>	<p>The leading dimension of the output array <i>q</i>.</p> <p>$ldq \geq \max(1, m)$ if <i>vect</i> = 'Q' or 'B', $ldq \geq 1$ otherwise.</p>
<i>ldpt</i>	<p>The leading dimension of the output array <i>pt</i>.</p> <p>$ldpt \geq \max(1, n)$ if <i>vect</i> = 'P' or 'B', $ldpt \geq 1$ otherwise.</p>
<i>ldc</i>	<p>The leading dimension of the array <i>c</i>.</p> <p>$ldc \geq \max(1, m)$ if $ncc > 0$; $ldc \geq 1$ if $ncc = 0$.</p>

Output Parameters

<i>ab</i>	Overwritten by values generated during the reduction.
<i>d</i>	Array, size at least $\max(1, \min(m, n))$. Contains the diagonal elements of the matrix <i>B</i> .
<i>e</i>	<p>Array, size at least $\max(1, \min(m, n) - 1)$.</p> <p>Contains the off-diagonal elements of <i>B</i>.</p>
<i>q, pt</i>	<p>Arrays:</p> <p><i>q</i>size $\max(1, ldq*m)$ contains the output <i>m</i>-by-<i>m</i> matrix <i>Q</i>.</p> <p><i>pt</i>size $\max(1, ldpt*n)$ contains the output <i>n</i>-by-<i>n</i> matrix <i>P</i>^T.</p>
<i>c</i>	<p>Overwritten by the product $Q^H * C$.</p> <p><i>c</i> is not referenced if $ncc = 0$.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrices *Q*, *B*, and *P* satisfy $Q^H B P^H = A + E$, where $\|E\|_2 = c(n)\varepsilon \|A\|_2$, $c(n)$ is a modestly increasing function of *n*, and ε is the machine precision.

If $m = n$, the total number of floating-point operations for real flavors is approximately the sum of:

$6*n^2*(kl + ku)$ if $vect = 'N'$ and $ncc = 0$,

$3*n^2*ncc*(kl + ku - 1)/(kl + ku)$ if C is updated, and

$3*n^3*(kl + ku - 1)/(kl + ku)$ if either Q or P^H is generated (double this if both).

To estimate the number of operations for complex flavors, use the same formulas with the coefficients 20 and 10 (instead of 6 and 3).

?orgbr

Generates the real orthogonal matrix Q or P^T determined by ?gebrd.

Syntax

```
lapack_int LAPACKE_sorgbr (int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int k, float* a, lapack_int lda, const float* tau);
```

```
lapack_int LAPACKE_dorgbr (int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int k, double* a, lapack_int lda, const double* tau);
```

Include Files

- mkl.h

Description

The routine generates the whole or part of the orthogonal matrices Q and P^T formed by the routines [gebrd](#). Use this routine after a call to [sgebrd](#)/[dgebrd](#). All valid combinations of arguments are described in *Input parameters*. In most cases you need the following:

To compute the whole m -by- m matrix Q :

```
LAPACKE_?orgbr(matrix_layout, 'Q', m, m, n, a, lda, tau )
```

(note that the array a must have at least m columns).

To form the n leading columns of Q if $m > n$:

```
LAPACKE_?orgbr(matrix_layout, 'Q', m, n, n, a, lda, tau )
```

To compute the whole n -by- n matrix P^T :

```
LAPACKE_?orgbr(matrix_layout, 'P', n, n, m, a, lda, tau )
```

(note that the array a must have at least n rows).

To form the m leading rows of P^T if $m < n$:

```
LAPACKE_?orgbr(matrix_layout, 'P', m, n, m, a, lda, tau )
```

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>vect</i>	Must be 'Q' or 'P'. If <i>vect</i> = 'Q', the routine generates the matrix Q . If <i>vect</i> = 'P', the routine generates the matrix P^T .

m, n	<p>The number of rows (m) and columns (n) in the matrix Q or P^T to be returned ($m \geq 0, n \geq 0$).</p> <p>If $vect = 'Q', m \geq n \geq \min(m, k)$.</p> <p>If $vect = 'P', n \geq m \geq \min(n, k)$.</p>
k	<p>If $vect = 'Q'$, the number of columns in the original m-by-k matrix reduced by gebrd.</p> <p>If $vect = 'P'$, the number of rows in the original k-by-n matrix reduced by gebrd.</p>
a	Array, size at least $lda*n$ for column major layout and $lda*m$ for row major layout. The vectors which define the elementary reflectors, as returned by gebrd .
lda	The leading dimension of the array a . $lda \geq \max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.
tau	<p>Array, size $\min(m, k)$ if $vect = 'Q'$, $\min(n, k)$ if $vect = 'P'$.</p> <p>Scalar factor of the elementary reflector $H(i)$ or $G(i)$, which determines Q and P^T as returned by gebrd in the array $tauq$ or $taup$.</p>

Output Parameters

a	Overwritten by the orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by $vect, m$, and n .
-----	--

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$.

The approximate numbers of floating-point operations for the cases listed in *Description* are as follows:

To form the whole of Q :

$$(4/3) * n * (3m^2 - 3m * n + n^2) \text{ if } m > n;$$

$$(4/3) * m^3 \text{ if } m \leq n.$$

To form the n leading columns of Q when $m > n$:

$$(2/3) * n^2 * (3m - n) \text{ if } m > n.$$

To form the whole of P^T :

$$(4/3) * n^3 \text{ if } m \geq n;$$

$$(4/3) * m * (3n^2 - 3m * n + m^2) \text{ if } m < n.$$

To form the m leading columns of P^T when $m < n$:

$$(2/3) * n^2 * (3m - n) \text{ if } m > n.$$

The complex counterpart of this routine is [ungbr](#).

?ormbr

Multiplies an arbitrary real matrix by the real orthogonal matrix Q or P^T determined by ?gebrd.

Syntax

```
lapack_int LAPACKE_sormbr (int matrix_layout, char vect, char side, char trans,
lapack_int m, lapack_int n, lapack_int k, const float* a, lapack_int lda, const float*
tau, float* c, lapack_int ldc);
```

```
lapack_int LAPACKE_dormbr (int matrix_layout, char vect, char side, char trans,
lapack_int m, lapack_int n, lapack_int k, const double* a, lapack_int lda, const
double* tau, double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

Given an arbitrary real matrix C , this routine forms one of the matrix products Q^*C , $Q^T C$, C^*Q , C^*Q^T , P^*C , $P^T C$, C^*P , C^*P^T , where Q and P are orthogonal matrices computed by a call to [gebrd](#). The routine overwrites the product on C .

Input Parameters

In the descriptions below, r denotes the order of Q or P^T :

If $side = 'L'$, $r = m$; if $side = 'R'$, $r = n$.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>vect</i>	Must be 'Q' or 'P'. If $vect = 'Q'$, then Q or Q^T is applied to C . If $vect = 'P'$, then P or P^T is applied to C .
<i>side</i>	Must be 'L' or 'R'. If $side = 'L'$, multipliers are applied to C from the left. If $side = 'R'$, they are applied to C from the right.
<i>trans</i>	Must be 'N' or 'T'. If $trans = 'N'$, then Q or P is applied to C . If $trans = 'T'$, then Q^T or P^T is applied to C .
<i>m</i>	The number of rows in C .
<i>n</i>	The number of columns in C .
<i>k</i>	One of the dimensions of A in ?gebrd: If $vect = 'Q'$, the number of columns in A ; If $vect = 'P'$, the number of rows in A . Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.

a, *c*

Arrays:

a is the array *a* as returned by ?gebrd.The size of *a* depends on the value of the *matrix_layout*, *vect*, and *side* parameters:

<i>matrix_layout</i>	<i>vect</i>	<i>side</i>	size
column major	'Q'	-	$\max(1, lda*k)$
column major	'P'	'L'	$\max(1, lda*m)$
column major	'P'	'R'	$\max(1, lda*n)$
row major	'Q'	'L'	$\max(1, lda*m)$
row major	'Q'	'R'	$\max(1, lda*n)$
row major	'P'	-	$\max(1, lda*k)$

c(size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout) holds the matrix *C*.*lda*The leading dimension of *a*. Constraints:*lda* $\geq \max(1, r)$ for column major layout and at least $\max(1, k)$ for row major layout if *vect* = 'Q';*lda* $\geq \max(1, \min(r, k))$ for column major layout and at least $\max(1, r)$ for row major layout if *vect* = 'P'.*ldc*The leading dimension of *c*; *ldc* $\geq \max(1, m)$ for column major layout and *ldc* $\geq \max(1, n)$ for row major layout .*tau*Array, size at least $\max(1, \min(r, k))$.For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd.

Output Parameters

*c*Overwritten by the product Q^*C , Q^T*C , $C*Q$, $C*Q^T$, P^*C , P^T*C , C^*P , or C^*P^T , as specified by *vect*, *side*, and *trans*.

Return Values

This function returns a value *info*.If *info*=0, the execution is successful.If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|C\|_2$.

The total number of floating-point operations is approximately

 $2*n*k(2*m - k)$ if *side* = 'L' and $m \geq k$; $2*m*k(2*n - k)$ if *side* = 'R' and $n \geq k$;

$2*m^2*n$ if $side = 'L'$ and $m < k$;

$2*n^2*m$ if $side = 'R'$ and $n < k$.

The complex counterpart of this routine is [unmbr](#).

?ungbr

Generates the complex unitary matrix Q or P^H determined by ?gebrd.

Syntax

```
lapack_int LAPACKE_cungbr (int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int k, lapack_complex_float* a, lapack_int lda, const lapack_complex_float*
tau);
```

```
lapack_int LAPACKE_zungbr (int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int k, lapack_complex_double* a, lapack_int lda, const lapack_complex_double*
tau);
```

Include Files

- mkl.h

Description

The routine generates the whole or part of the unitary matrices Q and P^H formed by the routines [gebrd](#). Use this routine after a call to [cgebrd/zgebrd](#). All valid combinations of arguments are described in *Input Parameters*; in most cases you need the following:

To compute the whole m -by- m matrix Q , use:

```
LAPACKE_?ungbr(matrix_layout, 'Q', m, m, n, a, lda, tau)
```

(note that the array a must have at least m columns).

To form the n leading columns of Q if $m > n$, use:

```
LAPACKE_?ungbr(matrix_layout, 'Q', m, n, n, a, lda, tau)
```

To compute the whole n -by- n matrix P^H , use:

```
LAPACKE_?ungbr(matrix_layout, 'P', n, n, m, a, lda, tau)
```

(note that the array a must have at least n rows).

To form the m leading rows of P^H if $m < n$, use:

```
LAPACKE_?ungbr(matrix_layout, 'P', m, m, n, a, lda, tau)
```

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>vect</i>	Must be 'Q' or 'P'. If <i>vect</i> = 'Q', the routine generates the matrix Q . If <i>vect</i> = 'P', the routine generates the matrix P^H .
<i>m</i>	The number of required rows of Q or P^H .
<i>n</i>	The number of required columns of Q or P^H .

<i>k</i>	<p>One of the dimensions of <i>A</i> in ?gebrd:</p> <p>If <i>vect</i> = 'Q', the number of columns in <i>A</i>;</p> <p>If <i>vect</i> = 'P', the number of rows in <i>A</i>.</p> <p>Constraints: $m \geq 0, n \geq 0, k \geq 0$.</p> <p>For <i>vect</i> = 'Q': $k \leq n \leq m$ if $m > k$, or $m = n$ if $m \leq k$.</p> <p>For <i>vect</i> = 'P': $k \leq m \leq n$ if $n > k$, or $m = n$ if $n \leq k$.</p>
<i>a</i>	<p>Arrays:</p> <p><i>a</i>, size at least $lda*n$ for column major layout and $lda*m$ for row major layout, is the array <i>a</i> as returned by ?gebrd.</p>
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
<i>tau</i>	<p>For <i>vect</i> = 'Q', the array <i>tauq</i> as returned by ?gebrd. For <i>vect</i> = 'P', the array <i>taup</i> as returned by ?gebrd.</p> <p>The dimension of <i>tau</i> must be at least $\max(1, \min(m, k))$ for <i>vect</i> = 'Q', or $\max(1, \min(m, k))$ for <i>vect</i> = 'P'.</p>

Output Parameters

<i>a</i>	Overwritten by the orthogonal matrix <i>Q</i> or P^T (or the leading rows or columns thereof) as specified by <i>vect</i> , <i>m</i> , and <i>n</i> .
----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon)$.

The approximate numbers of possible floating-point operations are listed below:

To compute the whole matrix *Q*:

$(16/3)n(3m^2 - 3m*n + n^2)$ if $m > n$;

$(16/3)m^3$ if $m \leq n$.

To form the *n* leading columns of *Q* when $m > n$:

$(8/3)n^2(3m - n^2)$.

To compute the whole matrix P^H :

$(16/3)n^3$ if $m \geq n$;

$(16/3)m(3n^2 - 3m*n + m^2)$ if $m < n$.

To form the *m* leading columns of P^H when $m < n$:

$(8/3)n^2(3m - n^2)$ if $m > n$.

The real counterpart of this routine is [orgbr](#).

?unmbr

Multiplies an arbitrary complex matrix by the unitary matrix Q or P determined by ?gebrd.

Syntax

```
lapack_int LAPACKE_cunmbr (int matrix_layout, char vect, char side, char trans,
lapack_int m, lapack_int n, lapack_int k, const lapack_complex_float* a, lapack_int
lda, const lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);

lapack_int LAPACKE_zunmbr (int matrix_layout, char vect, char side, char trans,
lapack_int m, lapack_int n, lapack_int k, const lapack_complex_double* a, lapack_int
lda, const lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

Given an arbitrary complex matrix C , this routine forms one of the matrix products $Q*C$, Q^H*C , $C*Q$, $C*Q^H$, $P*C$, P^H*C , $C*P$, or $C*P^H$, where Q and P are unitary matrices computed by a call to [gebrd/gebrd](#). The routine overwrites the product on C .

Input Parameters

In the descriptions below, r denotes the order of Q or P^H :

If $side = 'L'$, $r = m$; if $side = 'R'$, $r = n$.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>vect</i>	Must be 'Q' or 'P'. If <i>vect</i> = 'Q', then Q or Q^H is applied to C . If <i>vect</i> = 'P', then P or P^H is applied to C .
<i>side</i>	Must be 'L' or 'R'. If <i>side</i> = 'L', multipliers are applied to C from the left. If <i>side</i> = 'R', they are applied to C from the right.
<i>trans</i>	Must be 'N' or 'C'. If <i>trans</i> = 'N', then Q or P is applied to C . If <i>trans</i> = 'C', then Q^H or P^H is applied to C .
<i>m</i>	The number of rows in C .
<i>n</i>	The number of columns in C .
<i>k</i>	One of the dimensions of A in ?gebrd: If <i>vect</i> = 'Q', the number of columns in A ; If <i>vect</i> = 'P', the number of rows in A . Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.

a, *c*

Arrays:

a is the array *a* as returned by ?gebrd.The size of *a* depends on the value of the *matrix_layout*, *vect*, and *side* parameters:

<i>matrix_layout</i>	<i>vect</i>	<i>side</i>	size
column major	'Q'	-	$\max(1, lda*k)$
column major	'P'	'L'	$\max(1, lda*m)$
column major	'P'	'R'	$\max(1, lda*n)$
row major	'Q'	'L'	$\max(1, lda*m)$
row major	'Q'	'R'	$\max(1, lda*n)$
row major	'P'	-	$\max(1, lda*k)$

c(size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout) holds the matrix *C*.*lda*The leading dimension of *a*. Constraints:*lda* $\geq \max(1, r)$ for column major layout and at least $\max(1, k)$ for row major layout if *vect* = 'Q';*lda* $\geq \max(1, \min(r, k))$ for column major layout and at least $\max(1, r)$ for row major layout if *vect* = 'P'.*ldc*The leading dimension of *c*; *ldc* $\geq \max(1, m)$.*tau*Array, size at least $\max(1, \min(r, k))$.For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd.

Output Parameters

*c*Overwritten by the product $Q*C$, Q^H*C , $C*Q$, $C*Q^H$, $P*C$, P^H*C , $C*P$, or $C*P^H$, as specified by *vect*, *side*, and *trans*.

Return Values

This function returns a value *info*.If *info*=0, the execution is successful.If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\varepsilon) * \|C\|_2$.

The total number of floating-point operations is approximately

 $8*n*k(2*m - k)$ if *side* = 'L' and $m \geq k$;

$8*m*k(2*n - k)$ if $side = 'R'$ and $n \geq k$;

$8*m^2*n$ if $side = 'L'$ and $m < k$;

$8*n^2*m$ if $side = 'R'$ and $n < k$.

The real counterpart of this routine is [ormbr](#).

?bdsqr

Computes the singular value decomposition of a general matrix that has been reduced to bidiagonal form.

Syntax

```
lapack_int LAPACKE_sbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncv,
lapack_int nru, lapack_int ncc, float* d, float* e, float* vt, lapack_int ldvt, float*
u, lapack_int ldu, float* c, lapack_int ldc );
```

```
lapack_int LAPACKE_dbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncv,
lapack_int nru, lapack_int ncc, double* d, double* e, double* vt, lapack_int ldvt,
double* u, lapack_int ldu, double* c, lapack_int ldc );
```

```
lapack_int LAPACKE_cbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncv,
lapack_int nru, lapack_int ncc, float* d, float* e, lapack_complex_float* vt,
lapack_int ldvt, lapack_complex_float* u, lapack_int ldu, lapack_complex_float* c,
lapack_int ldc );
```

```
lapack_int LAPACKE_zbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncv,
lapack_int nru, lapack_int ncc, double* d, double* e, lapack_complex_double* vt,
lapack_int ldvt, lapack_complex_double* u, lapack_int ldu, lapack_complex_double* c,
lapack_int ldc );
```

Include Files

- `mk1.h`

Description

The routine computes the singular values and, optionally, the right and/or left singular vectors from the [Singular Value Decomposition](#) (SVD) of a real n -by- n (upper or lower) bidiagonal matrix B using the implicit zero-shift QR algorithm. The SVD of B has the form $B = Q*S*P^H$ where S is the diagonal matrix of singular values, Q is an orthogonal matrix of left singular vectors, and P is an orthogonal matrix of right singular vectors. If left singular vectors are requested, this subroutine actually returns $U*Q$ instead of Q , and, if right singular vectors are requested, this subroutine returns P^H*VT instead of P^H , for given real/complex input matrices U and VT . When U and VT are the orthogonal/unitary matrices that reduce a general matrix A to bidiagonal form: $A = U*B*VT$, as computed by [?gebrd](#), then

$$A = (U*Q) * S * (P^H*VT)$$

is the SVD of A . Optionally, the subroutine may also compute Q^H*C for a given real/complex input matrix C .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Must be 'U' or 'L'.

	<p>If <code>uplo = 'U'</code>, B is an upper bidiagonal matrix.</p> <p>If <code>uplo = 'L'</code>, B is a lower bidiagonal matrix.</p>
n	The order of the matrix B ($n \geq 0$).
$ncvt$	<p>The number of columns of the matrix VT, that is, the number of right singular vectors ($ncvt \geq 0$).</p> <p>Set $ncvt = 0$ if no right singular vectors are required.</p>
nru	<p>The number of rows in U, that is, the number of left singular vectors ($nru \geq 0$).</p> <p>Set $nru = 0$ if no left singular vectors are required.</p>
ncc	The number of columns in the matrix C used for computing the product $Q^H * C$ ($ncc \geq 0$). Set $ncc = 0$ if no matrix C is supplied.
d, e	<p>Arrays:</p> <p>d contains the diagonal elements of B.</p> <p>The size of d must be at least $\max(1, n)$.</p> <p>e contains the $(n-1)$ off-diagonal elements of B.</p> <p>The size of e must be at least $\max(1, n - 1)$.</p>
vt, u, c	<p>Arrays:</p> <p>vt, size $\max(1, ldvt * ncvt)$ for column major layout and $\max(1, ldvt * n)$ for row major layout, contains an n-by-$ncvt$ matrix VT.</p> <p>vt is not referenced if $ncvt = 0$.</p> <p>u, size $\max(1, ldu * n)$ for column major layout and $\max(1, ldu * nru)$ for row major layout, contains an nru by n matrix U.</p> <p>u is not referenced if $nru = 0$.</p> <p>c, size $\max(1, ldc * ncc)$ for column major layout and $\max(1, ldc * n)$ for row major layout, contains the n-by-ncc matrix C for computing the product $Q^H * C$.</p>
$ldvt$	<p>The leading dimension of vt. Constraints:</p> <p>$ldvt \geq \max(1, n)$ if $ncvt > 0$ for column major layout and $ldvt \geq \max(1, ncvt)$ for row major layout;</p> <p>$ldvt \geq 1$ if $ncvt = 0$.</p>
ldu	<p>The leading dimension of u. Constraint:</p> <p>$ldu \geq \max(1, nru)$ for column major layout and $ldu \geq \max(1, n)$ for row major layout .</p>
ldc	<p>The leading dimension of c. Constraints:</p> <p>$ldc \geq \max(1, n)$ if $ncc > 0$ for column major layout and $ldc \geq \max(1, ncc)$ for row major layout; $ldc \geq 1$ otherwise.</p>

Output Parameters

d	On exit, if $info = 0$, overwritten by the singular values in decreasing order (see $info$).
e	On exit, if $info = 0$, e is destroyed. See also $info$ below.
c	Overwritten by the product $Q^H * C$.
vt	On exit, this array is overwritten by $P^H * VT$. Not referenced if $ncvt = 0$.
u	On exit, this array is overwritten by $U * Q$. Not referenced if $nru = 0$.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info > 0$,

If $ncvt = nru = ncc = 0$,

- $info = 1$, a split was marked by a positive value in e
- $info = 2$, the current block of z not diagonalized after $100*n$ iterations (in the inner `while` loop)
- $info = 3$, termination criterion of the outer `while` loop is not met (the program created more than n unreduced blocks).

In all other cases when $ncvt$, nru , or $ncc > 0$, the algorithm did not converge; d and e contain the elements of a bidiagonal matrix that is orthogonally similar to the input matrix B ; if $info = i$, i elements of e have not converged to zero.

Application Notes

Each singular value and singular vector is computed to high relative accuracy. However, the reduction to bidiagonal form (prior to calling the routine) may decrease the relative accuracy in the small singular values of the original matrix if its singular values vary widely in magnitude.

If s_i is an exact singular value of B , and s_i is the corresponding computed value, then

$$|s_i - \sigma_i| \leq p^*(m, n) * \varepsilon * \sigma_i$$

where $p(m, n)$ is a modestly increasing function of m and n , and ε is the machine precision.

If only singular values are computed, they are computed more accurately than when some singular vectors are also computed (that is, the function $p(m, n)$ is smaller).

If u_i is the corresponding exact left singular vector of B , and w_i is the corresponding computed left singular vector, then the angle $\theta(u_i, w_i)$ between them is bounded as follows:

$$\theta(u_i, w_i) \leq p(m, n) * \varepsilon / \min_{i \neq j} (|\sigma_i - \sigma_j| / |\sigma_i + \sigma_j|).$$

Here $\min_{i \neq j} (|\sigma_i - \sigma_j| / |\sigma_i + \sigma_j|)$ is the *relative gap* between σ_i and the other singular values. A similar error bound holds for the right singular vectors.

The total number of real floating-point operations is roughly proportional to n^2 if only the singular values are computed. About $6n^2 * nru$ additional operations ($12n^2 * nru$ for complex flavors) are required to compute the left singular vectors and about $6n^2 * ncvt$ operations ($12n^2 * ncvt$ for complex flavors) to compute the right singular vectors.

?bdsdc

Computes the singular value decomposition of a real bidiagonal matrix using a divide and conquer method.

Syntax

```
lapack_int LAPACKE_sbdsdc (int matrix_layout, char uplo, char compq, lapack_int n,
float* d, float* e, float* u, lapack_int ldu, float* vt, lapack_int ldvt, float* q,
lapack_int* iq);
```

```
lapack_int LAPACKE_dbdsdc (int matrix_layout, char uplo, char compq, lapack_int n,
double* d, double* e, double* u, lapack_int ldu, double* vt, lapack_int ldvt, double* q,
lapack_int* iq);
```

Include Files

- mkl.h

Description

The routine computes the [Singular Value Decomposition](#) (SVD) of a real n -by- n (upper or lower) bidiagonal matrix B : $B = U \Sigma V^T$, using a divide and conquer method, where Σ is a diagonal matrix with non-negative diagonal elements (the singular values of B), and U and V are orthogonal matrices of left and right singular vectors, respectively. `?bdsdc` can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

This routine

uses `?lasd0`, `?lasd1`, `?lasd2`, `?lasd3`, `?lasd4`, `?lasd5`, `?lasd6`, `?lasd7`, `?lasd8`, `?lasd9`, `?lasda`, `?lasdq`, `?lasdt`.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Must be 'U' or 'L'. If <code>uplo = 'U'</code> , B is an upper bidiagonal matrix. If <code>uplo = 'L'</code> , B is a lower bidiagonal matrix.
<code>compq</code>	Must be 'N', 'P', or 'I'. If <code>compq = 'N'</code> , compute singular values only. If <code>compq = 'P'</code> , compute singular values and compute singular vectors in compact form. If <code>compq = 'I'</code> , compute singular values and singular vectors.
<code>n</code>	The order of the matrix B ($n \geq 0$).
<code>d, e</code>	Arrays: d contains the n diagonal elements of the bidiagonal matrix B . The size of d must be at least $\max(1, n)$. e contains the off-diagonal elements of the bidiagonal matrix B . The size of e must be at least $\max(1, n)$.
<code>ldu</code>	The leading dimension of the output array u ; $ldu \geq 1$.

If singular vectors are desired, then $ldu \geq \max(1, n)$, regardless of the value of `matrix_layout`.

`ldvt`

The leading dimension of the output array `vt`; $ldvt \geq 1$.

If singular vectors are desired, then $ldvt \geq \max(1, n)$, regardless of the value of `matrix_layout`.

Output Parameters

`d`

If `info = 0`, overwritten by the singular values of B .

`e`

On exit, `e` is overwritten.

`u, vt, q`

Arrays: `u`(size $ldu \times n$), `vt`(size $ldvt \times n$), `q`(size $\geq n \times (11 + 2 \times smlsiz + 8 \times \text{int}(\log_2(n/(smlsiz+1))))$) where `smlsiz` is returned by `ilaenv` and is equal to maximum size of the subproblems at the bottom of the computation tree)..

If `compq = 'I'`, then on exit `u` contains the left singular vectors of the bidiagonal matrix B , unless `info` $\neq 0$ (see `info`). For other values of `compq`, `u` is not referenced.

if `compq = 'I'`, then on exit `vtT` contains the right singular vectors of the bidiagonal matrix B , unless `info` $\neq 0$ (see `info`). For other values of `compq`, `vt` is not referenced.

If `compq = 'P'`, then on exit, if `info = 0`, `q` and `iq` contain the left and right singular vectors in a compact form. Specifically, `q` contains all the `float` (for `sbdsc`) or `double` (for `dbdsc`) data for singular vectors. For other values of `compq`, `q` is not referenced.

`iq`

Array: `iq`(size $\geq n \times (3 + 3 \times \text{int}(\log_2(n/(smlsiz+1))))$) where `smlsiz` is returned by `ilaenv` and is equal to maximum size of the subproblems at the bottom of the computation tree.).

If `compq = 'P'`, then on exit, if `info = 0`, `q` and `iq` contain the left and right singular vectors in a compact form. Specifically, `iq` contains all the `lapack_int` data for singular vectors. For other values of `compq`, `iq` is not referenced.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i`, the algorithm failed to compute a singular value. The update process of divide and conquer failed.

Symmetric Eigenvalue Problems: LAPACK Computational Routines

Symmetric eigenvalue problems are posed as follows: given an n -by- n real symmetric or complex Hermitian matrix A , find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \text{ (or, equivalently, } z^H A = \lambda z^H \text{)}.$$

In such eigenvalue problems, all n eigenvalues are real not only for real symmetric but also for complex Hermitian matrices A , and there exists an orthonormal system of n eigenvectors. If A is a symmetric or Hermitian positive-definite matrix, all eigenvalues are positive.

To solve a symmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to tridiagonal form and then solve the eigenvalue problem with the tridiagonal matrix obtained. LAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in [Table "Computational Routines for Solving Symmetric Eigenvalue Problems"](#).

There are different routines for symmetric eigenvalue problems, depending on whether you need all eigenvectors or only some of them or eigenvalues only, whether the matrix A is positive-definite or not, and so on.

These routines are based on three primary algorithms for computing eigenvalues and eigenvectors of symmetric problems: the divide and conquer algorithm, the QR algorithm, and bisection followed by inverse iteration. The divide and conquer algorithm is generally more efficient and is recommended for computing all eigenvalues and eigenvectors. Furthermore, to solve an eigenvalue problem using the divide and conquer algorithm, you need to call only one routine. In general, more than one routine has to be called if the QR algorithm or bisection followed by inverse iteration is used.

Computational Routines for Solving Symmetric Eigenvalue Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to tridiagonal form $A = QTQ^H$ (full storage)	sytrd	hetrd
Reduce to tridiagonal form $A = QTQ^H$ (packed storage)	sptdr	hptrd
Reduce to tridiagonal form $A = QTQ^H$ (band storage).	sbtrd	hbtrd
Generate matrix Q (full storage)	orgtr	ungtr
Generate matrix Q (packed storage)	opgtr	upgtr
Apply matrix Q (full storage)	ormtr	unmtr
Apply matrix Q (packed storage)	opmtr	upmtr
Find all eigenvalues of a tridiagonal matrix T	sterf	
Find all eigenvalues and eigenvectors of a tridiagonal matrix T	steqr stedc	steqr stedc
Find all eigenvalues and eigenvectors of a tridiagonal positive-definite matrix T .	pteqr	pteqr
Find selected eigenvalues of a tridiagonal matrix T	stebz stegr	stegr
Find selected eigenvectors of a tridiagonal matrix T	stein stegr	stein stegr
Find selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix T	stemr	stemr
Compute the reciprocal condition numbers for the eigenvectors	disna	disna

*?sytrd**Reduces a real symmetric matrix to tridiagonal form.*

Syntax

```
lapack_int LAPACKE_ssytrd (int matrix_layout, char uplo, lapack_int n, float* a,
lapack_int lda, float* d, float* e, float* tau);
```

```
lapack_int LAPACKE_dsytrd (int matrix_layout, char uplo, lapack_int n, double* a,
lapack_int lda, double* d, double* e, double* tau);
```

Include Files

- mkl.h

Description

The routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q^T T Q$. The orthogonal matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation (see *Application Notes* below).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of A .
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>a</i>	<i>a</i> (size $\max(1, lda \cdot n)$) is an array containing either upper or lower triangular part of the matrix A , as specified by <i>uplo</i> . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of <i>a</i> contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of <i>a</i> contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit, if <i>uplo</i> = 'U', the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements above the first superdiagonal, with the array <i>tau</i> , represent the orthogonal matrix Q as a product of elementary reflectors; if <i>uplo</i> = 'L', the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements below the first subdiagonal, with the array <i>tau</i> , represent the orthogonal matrix Q as a product of elementary reflectors.
----------	--

d, *e*, *tau*

Arrays:

d contains the diagonal elements of the matrix *T*.

The size of *d* must be at least $\max(1, n)$.

e contains the off-diagonal elements of *T*.

The size of *e* must be at least $\max(1, n-1)$.

tau stores $(n-1)$ scalars that define elementary reflectors in decomposition of the orthogonal matrix *Q* in a product of $n-1$ elementary reflectors. *tau*(*n*) is used as workspace.

The size of *tau* must be at least $\max(1, n)$.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrix *T* is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of *n*, and ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

[orgtr](#) to form the computed matrix *Q* explicitly

[ormtr](#) to multiply a real matrix by *Q*.

The complex counterpart of this routine is [?hetrd](#).

?orgtr

Generates the real orthogonal matrix Q determined by ?sytrd.

Syntax

```
lapack_int LAPACKE_sorgtr (int matrix_layout, char uplo, lapack_int n, float* a,
lapack_int lda, const float* tau);
```

```
lapack_int LAPACKE_dorgtr (int matrix_layout, char uplo, lapack_int n, double* a,
lapack_int lda, const double* tau);
```

Include Files

- `mkl.h`

Description

The routine explicitly generates the *n*-by-*n* orthogonal matrix *Q* formed by *?sytrd* when reducing a real symmetric matrix *A* to tridiagonal form: $A = Q^T T Q^T$. Use this routine after a call to *?sytrd*.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sytrd.
<i>n</i>	The order of the matrix Q ($n \geq 0$).
<i>a</i> , <i>tau</i>	Arrays: <i>a</i> (size $\max(1, lda*n)$) is the array <i>a</i> as returned by ?sytrd. <i>tau</i> is the array <i>tau</i> as returned by ?sytrd. The size of <i>tau</i> must be at least $\max(1, n-1)$.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	Overwritten by the orthogonal matrix Q .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is [ungtr](#).

?ormtr

Multiplies a real matrix by the real orthogonal matrix Q determined by ?sytrd.

Syntax

```
lapack_int LAPACKE_sormtr (int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const float* a, lapack_int lda, const float* tau, float* c,
lapack_int ldc);
```

```
lapack_int LAPACKE_dormtr (int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const double* a, lapack_int lda, const double* tau, double*
c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q formed by `sytrd` when reducing a real symmetric matrix A to tridiagonal form: $A = Q^T Q$. Use this routine after a call to `sytrd`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q^T C$, $Q^T C$, $C^T Q$, or $C^T Q^T$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>uplo</i>	Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to <code>sytrd</code> .
<i>trans</i>	Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>a</i> , <i>c</i> , <i>tau</i>	<i>a</i> (size $\max(1, lda * r)$) and <i>tau</i> are the arrays returned by <code>sytrd</code> . The size of <i>tau</i> must be at least $\max(1, r - 1)$. <i>c</i> (size $\max(1, ldc * n)$ for column major layout and $\max(1, ldc * m)$ for row major layout) contains the matrix C .
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, r)$.
<i>ldc</i>	The leading dimension of <i>c</i> ; $ldc \geq \max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.

Output Parameters

<i>c</i>	Overwritten by the product $Q^T C$, $Q^T C$, $C^T Q$, or $C^T Q^T$ (as specified by <i>side</i> and <i>trans</i>).
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\epsilon) * \|C\|_2$.

The total number of floating-point operations is approximately $2*m^2*n$, if *side* = 'L', or $2*n^2*m$, if *side* = 'R'.

The complex counterpart of this routine is [unmtr](#).

?hetrd

Reduces a complex Hermitian matrix to tridiagonal form.

Syntax

```
lapack_int LAPACKE_chetrd( int matrix_layout, char uplo, lapack_int n,
lapack_complex_float* a, lapack_int lda, float* d, float* e, lapack_complex_float*
tau );
```

```
lapack_int LAPACKE_zhetrd( int matrix_layout, char uplo, lapack_int n,
lapack_complex_double* a, lapack_int lda, double* d, double* e, lapack_complex_double*
tau );
```

Include Files

- mkl.h

Description

The routine reduces a complex Hermitian matrix *A* to symmetric tridiagonal form *T* by a unitary similarity transformation: $A = Q*T*Q^H$. The unitary matrix *Q* is not formed explicitly but is represented as a product of *n*-1 elementary reflectors. Routines are provided to work with *Q* in this representation. (They are described later in this topic.)

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	<i>a</i> (size $\max(1, lda*n)$) is an array containing either upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>A</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>A</i> is not referenced.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit,
----------	----------

if `uplo = 'U'`, the diagonal and first superdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements above the first superdiagonal, with the array `tau`, represent the orthogonal matrix Q as a product of elementary reflectors;

if `uplo = 'L'`, the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements below the first subdiagonal, with the array `tau`, represent the orthogonal matrix Q as a product of elementary reflectors.

`d, e`

Arrays:

`d` contains the diagonal elements of the matrix T .

The dimension of `d` must be at least $\max(1, n)$.

`e` contains the off-diagonal elements of T .

The dimension of `e` must be at least $\max(1, n-1)$.

`tau`

Array, size at least $\max(1, n-1)$. Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of $n-1$ elementary reflectors.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

The computed matrix T is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

After calling this routine, you can call the following:

`ungtr` to form the computed matrix Q explicitly

`unmtr` to multiply a complex matrix by Q .

The real counterpart of this routine is [?sytrd](#).

[?ungtr](#)

Generates the complex unitary matrix Q determined by [?hetrd](#).

Syntax

```
lapack_int LAPACKC_cungtr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);

lapack_int LAPACKC_zungtr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);
```

Include Files

- `mk1.h`

Description

The routine explicitly generates the n -by- n unitary matrix Q formed by `?hetrd` when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q^*T^*Q^H$. Use this routine after a call to `?hetrd`.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Must be 'U' or 'L'. Use the same <code>uplo</code> as supplied to <code>?hetrd</code> .
<code>n</code>	The order of the matrix Q ($n \geq 0$).
<code>a, tau</code>	Arrays: <code>a</code> (size $\max(1, lda*n)$) is the array <code>a</code> as returned by <code>?hetrd</code> . <code>tau</code> is the array <code>tau</code> as returned by <code>?hetrd</code> . The dimension of <code>tau</code> must be at least $\max(1, n-1)$.
<code>lda</code>	The leading dimension of <code>a</code> ; at least $\max(1, n)$.

Output Parameters

<code>a</code>	Overwritten by the unitary matrix Q .
----------------	---

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

The computed matrix Q differs from an exactly unitary matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is [orgtr](#).

`?unmtr`

Multiplies a complex matrix by the complex unitary matrix Q determined by `?hetrd`.

Syntax

```
lapack_int LAPACKE_cunmtr (int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);
```

```
lapack_int LAPACKE_zunmtr (int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- `mk1.h`

Description

The routine multiplies a complex matrix C by Q or Q^H , where Q is the unitary matrix Q formed by `?hetrd` when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q^*T^*Q^H$. Use this routine after a call to `?hetrd`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>uplo</i>	Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to <code>?hetrd</code> .
<i>trans</i>	Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>a</i> , <i>c</i> , <i>tau</i>	<i>a</i> (size $\max(1, lda*r)$) and <i>tau</i> are the arrays returned by <code>?hetrd</code> . The dimension of <i>tau</i> must be at least $\max(1, r-1)$. <i>c</i> (size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout) contains the matrix C .
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, r)$.
<i>ldc</i>	The leading dimension of <i>c</i> ; $ldc \geq \max(1, n)$ for column major layout and $ldc \geq \max(1, m)$ for row major layout.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (as specified by <i>side</i> and <i>trans</i>).
----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\varepsilon) * \|C\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if *side* = 'L' or $8*n^2*m$ if *side* = 'R'.

The real counterpart of this routine is [ormtr](#).

?sptd

Reduces a real symmetric matrix to tridiagonal form using packed storage.

Syntax

```
lapack_int LAPACK_essptd (int matrix_layout, char uplo, lapack_int n, float* ap,
float* d, float* e, float* tau);
```

```
lapack_int LAPACK_dsptd (int matrix_layout, char uplo, lapack_int n, double* ap,
double* d, double* e, double* tau);
```

Include Files

- mkl.h

Description

The routine reduces a packed real symmetric matrix *A* to symmetric tridiagonal form *T* by an orthogonal similarity transformation: $A = Q^T T Q$. The orthogonal matrix *Q* is not formed explicitly but is represented as a product of *n*-1 elementary reflectors. Routines are provided for working with *Q* in this representation. See *Application Notes* below for details.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangle of <i>A</i> . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangle of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>ap</i>	Array, size at least $\max(1, n(n+1)/2)$. Contains either upper or lower triangle of <i>A</i> (as specified by <i>uplo</i>) in the packed form described in Matrix Storage Schemes .

Output Parameters

<i>ap</i>	Overwritten by the tridiagonal matrix <i>T</i> and details of the orthogonal matrix <i>Q</i> , as specified by <i>uplo</i> .
<i>d</i> , <i>e</i> , <i>tau</i>	<p>Arrays:</p> <p><i>d</i> contains the diagonal elements of the matrix <i>T</i>. The dimension of <i>d</i> must be at least $\max(1, n)$.</p> <p><i>e</i> contains the off-diagonal elements of <i>T</i>. The dimension of <i>e</i> must be at least $\max(1, n-1)$.</p> <p><i>tau</i> Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the matrix <i>Q</i> in a product of $n-1$ reflectors. The dimension of <i>tau</i> must be at least $\max(1, n-1)$.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The matrix *Q* is represented as a product of $n-1$ *elementary reflectors*, as follows :

- If *uplo* = 'U', $Q = H(n-1) \dots H(2)H(1)$

Each *H*(*i*) has the form

$$H(i) = I - \tau v v^T$$

where *tau* is a real scalar and *v* is a real vector with $v(i+1:n) = 0$ and $v(i) = 1$.

On exit, *tau* is stored in *tau*[*i* - 1], and $v(1:i-1)$ is stored in *AP*, overwriting $A(1:i-1, i+1)$.

- If *uplo* = 'L', $Q = H(1)H(2) \dots H(n-1)$

Each *H*(*i*) has the form

$$H(i) = I - \tau v v^T$$

where *tau* is a real scalar and *v* is a real vector with $v(1:i) = 0$ and $v(i+1) = 1$.

On exit, *tau* is stored in *tau*[*i* - 1], and $v(i+2:n)$ is stored in *AP*, overwriting $A(i+2:n, i)$.

The computed matrix *T* is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of *n*, and ϵ is the machine precision. The approximate number of floating-point operations is $(4/3) n^3$.

After calling this routine, you can call the following:

opgtr	to form the computed matrix <i>Q</i> explicitly
opmtr	to multiply a real matrix by <i>Q</i> .

The complex counterpart of this routine is [hptrd](#).

?opgtr

Generates the real orthogonal matrix Q determined by ?sptrd.

Syntax

```
lapack_int LAPACKE_sopgtr (int matrix_layout, char uplo, lapack_int n, const float* ap,
const float* tau, float* q, lapack_int ldq);
```

```
lapack_int LAPACKE_dopgtr (int matrix_layout, char uplo, lapack_int n, const double*
ap, const double* tau, double* q, lapack_int ldq);
```

Include Files

- mkl.h

Description

The routine explicitly generates the n -by- n orthogonal matrix Q formed by [sptrd](#) when reducing a packed real symmetric matrix A to tridiagonal form: $A = Q^T Q^T$. Use this routine after a call to ?sptrd.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sptrd.
<i>n</i>	The order of the matrix Q ($n \geq 0$).
<i>ap, tau</i>	Arrays <i>ap</i> and <i>tau</i> , as returned by ?sptrd. The size of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The size of <i>tau</i> must be at least $\max(1, n-1)$.
<i>ldq</i>	The leading dimension of the output array <i>q</i> ; at least $\max(1, n)$.

Output Parameters

<i>q</i>	Array, size (size $\max(1, ldq*n)$) . Contains the computed matrix Q .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is [upgtr](#).

?opmtr

Multiplies a real matrix by the real orthogonal matrix Q determined by ?sptrd.

Syntax

```
lapack_int LAPACKE_sopmtr (int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const float* ap, const float* tau, float* c, lapack_int
ldc);
```

```
lapack_int LAPACKE_dopmtr (int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const double* ap, const double* tau, double* c, lapack_int
ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q formed by [sptrd](#) when reducing a packed real symmetric matrix A to tridiagonal form: $A = Q^* T^* Q^T$. Use this routine after a call to ?sptrd.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q^* C$, $Q^T C$, $C^* Q$, or $C^* Q^T$ (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>uplo</i>	Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sptrd.
<i>trans</i>	Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>ap, tau, c</i>	<i>ap</i> and <i>tau</i> are the arrays returned by ?sptrd. The dimension of <i>ap</i> must be at least $\max(1, r(r+1)/2)$. The dimension of <i>tau</i> must be at least $\max(1, r-1)$.

c (size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout) contains the matrix C .

ldc

The leading dimension of c ; $ldc \geq \max(1, n)$ for column major layout and $ldc \geq \max(1, m)$ for row major layout.

Output Parameters

c

Overwritten by the product Q^*C , Q^T*C , $C*Q$, or $C*Q^T$ (as specified by *side* and *trans*).

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\epsilon) \|C\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $2*m^2*n$ if *side* = 'L', or $2*n^2*m$ if *side* = 'R'.

The complex counterpart of this routine is [upmtr](#).

?hptrd

Reduces a complex Hermitian matrix to tridiagonal form using packed storage.

Syntax

```
lapack_int LAPACKE_chptrd( int matrix_layout, char uplo, lapack_int n,
lapack_complex_float* ap, float* d, float* e, lapack_complex_float* tau );

lapack_int LAPACKE_zhptrd( int matrix_layout, char uplo, lapack_int n,
lapack_complex_double* ap, double* d, double* e, lapack_complex_double* tau );
```

Include Files

- mkl.h

Description

The routine reduces a packed complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q^*T^*Q^H$. The unitary matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation (see *Application Notes* below).

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo

Must be 'U' or 'L'.

If `uplo = 'U'`, `ap` stores the packed upper triangle of A .

If `uplo = 'L'`, `ap` stores the packed lower triangle of A .

`n` The order of the matrix A ($n \geq 0$).

`ap` Array, size at least $\max(1, n(n+1)/2)$. Contains either upper or lower triangle of A (as specified by `uplo`) in the packed form described in "[Matrix Storage Schemes](#)".

Output Parameters

`ap` Overwritten by the tridiagonal matrix T and details of the unitary matrix Q , as specified by `uplo`.

`d, e` Arrays:
`d` contains the diagonal elements of the matrix T .
 The size of `d` must be at least $\max(1, n)$.
`e` contains the off-diagonal elements of T .
 The size of `e` must be at least $\max(1, n-1)$.

`tau` Array, size at least $\max(1, n-1)$. Stores $(n-1)$ scalars that define elementary reflectors in decomposition of the unitary matrix Q in a product of reflectors.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

The computed matrix T is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

After calling this routine, you can call the following:

`upgtr` to form the computed matrix Q explicitly

`upmtr` to multiply a complex matrix by Q .

The real counterpart of this routine is [sptd](#).

?upgtr

Generates the complex unitary matrix Q determined by ?hptrd.

Syntax

```
lapack_int LAPACKE_cupgtr (int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* ap, const lapack_complex_float* tau, lapack_complex_float* q,
lapack_int ldq);
```

```
lapack_int LAPACKE_zupgtr (int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* ap, const lapack_complex_double* tau, lapack_complex_double* q,
lapack_int ldq);
```

Include Files

- mkl.h

Description

The routine explicitly generates the n -by- n unitary matrix Q formed by [hptrd](#) when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q^*T^*Q^H$. Use this routine after a call to [?hptrd](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?hptrd .
<i>n</i>	The order of the matrix Q ($n \geq 0$).
<i>ap, tau</i>	Arrays <i>ap</i> and <i>tau</i> , as returned by ?hptrd . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The dimension of <i>tau</i> must be at least $\max(1, n-1)$.
<i>ldq</i>	The leading dimension of the output array <i>q</i> ; at least $\max(1, n)$.

Output Parameters

<i>q</i>	Array, size (size $\max(1, ldq*n)$) . Contains the computed matrix Q .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is [opgtr](#).

[?upmtr](#)

Multiplies a complex matrix by the unitary matrix Q determined by [?hptrd](#).

Syntax

```
lapack_int LAPACKE_cupmtr (int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const lapack_complex_float* ap, const lapack_complex_float*
tau, lapack_complex_float* c, lapack_int ldc);
```

```
lapack_int LAPACKE_zupmtr (int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const lapack_complex_double* ap, const
lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a complex matrix C by Q or Q^H , where Q is the unitary matrix formed by [hptrd](#) when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q^*T^*Q^H$. Use this routine after a call to `?hptrd`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^H^*C , C^*Q , or C^*Q^H (overwriting the result on C).

Input Parameters

In the descriptions below, r denotes the order of Q :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>uplo</i>	Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to <code>?hptrd</code> .
<i>trans</i>	Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^H .
<i>m</i>	The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	The number of columns in C ($n \geq 0$).
<i>ap</i> , <i>tau</i> , <i>c</i> ,	<i>ap</i> and <i>tau</i> are the arrays returned by <code>?hptrd</code> . The size of <i>ap</i> must be at least $\max(1, r(r+1)/2)$. The size of <i>tau</i> must be at least $\max(1, r-1)$. $c(\text{size } \max(1, ldc*n) \text{ for column major layout and } \max(1, ldc*m) \text{ for row major layout})$ contains the matrix C .

ldc The leading dimension of *c*; $ldc \geq \max(1, m)$ for column major layout and $ldc \geq \max(1, n)$ for row major layout .

Output Parameters

c Overwritten by the product Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (as specified by *side* and *trans*).

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\varepsilon) * \|C\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if *side* = 'L' or $8*n^2*m$ if *side* = 'R'.

The real counterpart of this routine is [opmtr](#).

?sbtrd

Reduces a real symmetric band matrix to tridiagonal form.

Syntax

```
lapack_int LAPACKE_ssbtrd (int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int kd, float* ab, lapack_int ldab, float* d, float* e, float* q, lapack_int
ldq);
```

```
lapack_int LAPACKE_dsbtrd (int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int kd, double* ab, lapack_int ldab, double* d, double* e, double* q, lapack_int
ldq);
```

Include Files

- `mkl.h`

Description

The routine reduces a real symmetric band matrix *A* to symmetric tridiagonal form *T* by an orthogonal similarity transformation: $A = Q^*T^*Q^T$. The orthogonal matrix *Q* is determined as a product of Givens rotations.

If required, the routine can also form the matrix *Q* explicitly.

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

vect Must be 'V', 'N', or 'U'.

	If <i>vect</i> = 'V', the routine returns the explicit matrix <i>Q</i> .
	If <i>vect</i> = 'N', the routine does not return <i>Q</i> .
	If <i>vect</i> = 'U', the routine updates matrix <i>X</i> by forming $X*Q$.
<i>uplo</i>	Must be 'U' or 'L'.
	If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> .
	If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab, q</i>	<i>ab</i> (size at least $\max(1, ldab*n)$ for column major layout and at least $\max(1, ldab*(kd+1))$ for row major layout) is an array containing either upper or lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. <i>q</i> (size $\max(1, ldq*n)$) is an array. If <i>vect</i> = 'U', the <i>q</i> array must contain an <i>n</i> -by- <i>n</i> matrix <i>X</i> . If <i>vect</i> = 'N' or 'V', the <i>q</i> parameter need not be set.
<i>ldab</i>	The leading dimension of <i>ab</i> ; at least <i>kd</i> +1 for column major layout and <i>n</i> for row major layout .
<i>ldq</i>	The leading dimension of <i>q</i> . Constraints: $ldq \geq \max(1, n)$ if <i>vect</i> = 'V' or 'U'; $ldq \geq 1$ if <i>vect</i> = 'N'.

Output Parameters

<i>ab</i>	On exit, the diagonal elements of the array <i>ab</i> are overwritten by the diagonal elements of the tridiagonal matrix <i>T</i> . If <i>kd</i> > 0, the elements on the first superdiagonal (if <i>uplo</i> = 'U') or the first subdiagonal (if <i>uplo</i> = 'L') are overwritten by the off-diagonal elements of <i>T</i> . The rest of <i>ab</i> is overwritten by values generated during the reduction.
<i>d, e, q</i>	Arrays: <i>d</i> contains the diagonal elements of the matrix <i>T</i> . The size of <i>d</i> must be at least $\max(1, n)$. <i>e</i> contains the off-diagonal elements of <i>T</i> . The size of <i>e</i> must be at least $\max(1, n-1)$. <i>q</i> is not referenced if <i>vect</i> = 'N'. If <i>vect</i> = 'V', <i>q</i> contains the <i>n</i> -by- <i>n</i> matrix <i>Q</i> . If <i>vect</i> = 'U', <i>q</i> contains the product $X*Q$.

Return Values

This function returns a value *info*.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

Application Notes

The computed matrix T is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision. The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$.

The total number of floating-point operations is approximately $6n^2 * kd$ if `vect = 'N'`, with $3n^3 * (kd-1) / kd$ additional operations if `vect = 'V'`.

The complex counterpart of this routine is [hbtrd](#).

?hbtrd

Reduces a complex Hermitian band matrix to tridiagonal form.

Syntax

```
lapack_int LAPACKE_chbtrd( int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int kd, lapack_complex_float* ab, lapack_int ldab, float* d, float* e,
lapack_complex_float* q, lapack_int ldq );
```

```
lapack_int LAPACKE_zhbtrd( int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int kd, lapack_complex_double* ab, lapack_int ldab, double* d, double* e,
lapack_complex_double* q, lapack_int ldq );
```

Include Files

- `mkl.h`

Description

The routine reduces a complex Hermitian band matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q * T * Q^H$. The unitary matrix Q is determined as a product of Givens rotations.

If required, the routine can also form the matrix Q explicitly.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>vect</code>	Must be 'V', 'N', or 'U'. If <code>vect = 'V'</code> , the routine returns the explicit matrix Q . If <code>vect = 'N'</code> , the routine does not return Q . If <code>vect = 'U'</code> , the routine updates matrix X by forming $Q * X$.
<code>uplo</code>	Must be 'U' or 'L'. If <code>uplo = 'U'</code> , <code>ab</code> stores the upper triangular part of A . If <code>uplo = 'L'</code> , <code>ab</code> stores the lower triangular part of A .
<code>n</code>	The order of the matrix A ($n \geq 0$).

<i>kd</i>	The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab</i>	<i>ab</i> (size at least $\max(1, ldab*n)$ for column major layout and at least $\max(1, ldab*(kd+1))$ for row major layout) is an array containing either upper or lower triangular part of the matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.
<i>q</i>	<i>q</i> (size $\max(1, ldq*n)$) is an array. If <i>vect</i> = 'U', the <i>q</i> array must contain an <i>n</i> -by- <i>n</i> matrix <i>X</i> . If <i>vect</i> = 'N' or 'V', the <i>q</i> parameter need not be set.'
<i>ldab</i>	The leading dimension of <i>ab</i> ; at least <i>kd</i> +1 for column major layout and <i>n</i> for row major layout.
<i>ldq</i>	The leading dimension of <i>q</i> . Constraints: $ldq \geq \max(1, n)$ if <i>vect</i> = 'V' or 'U'; $ldq \geq 1$ if <i>vect</i> = 'N'.

Output Parameters

<i>ab</i>	On exit, the diagonal elements of the array <i>ab</i> are overwritten by the diagonal elements of the tridiagonal matrix <i>T</i> . If <i>kd</i> > 0, the elements on the first superdiagonal (if <i>uplo</i> = 'U') or the first subdiagonal (if <i>uplo</i> = 'L') are overwritten by the off-diagonal elements of <i>T</i> . The rest of <i>ab</i> is overwritten by values generated during the reduction.
<i>d</i> , <i>e</i>	Arrays: <i>d</i> contains the diagonal elements of the matrix <i>T</i> . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> contains the off-diagonal elements of <i>T</i> . The dimension of <i>e</i> must be at least $\max(1, n-1)$.
<i>q</i>	If <i>vect</i> = 'N', <i>q</i> is not referenced. If <i>vect</i> = 'V', <i>q</i> contains the <i>n</i> -by- <i>n</i> matrix <i>Q</i> . If <i>vect</i> = 'U', <i>q</i> contains the product $X^* Q$.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrix *T* is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n)*\epsilon*\|A\|_2$, $c(n)$ is a modestly increasing function of *n*, and ϵ is the machine precision. The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon)$.

The total number of floating-point operations is approximately $20n^2 \cdot kd$ if `vect = 'N'`, with $10n^3 \cdot (kd-1) / kd$ additional operations if `vect = 'V'`.

The real counterpart of this routine is [sbtrd](#).

?sterf

Computes all eigenvalues of a real symmetric tridiagonal matrix using QR algorithm.

Syntax

```
lapack_int LAPACKE_ssterf (lapack_int n, float* d, float* e);
lapack_int LAPACKE_dsterf (lapack_int n, double* d, double* e);
```

Include Files

- `mkl.h`

Description

The routine computes all the eigenvalues of a real symmetric tridiagonal matrix T (which can be obtained by reducing a symmetric or Hermitian matrix to tridiagonal form). The routine uses a square-root-free variant of the QR algorithm.

If you need not only the eigenvalues but also the eigenvectors, call [steqr](#).

Input Parameters

n	The order of the matrix T ($n \geq 0$).
d, e	Arrays: d contains the diagonal elements of T . The dimension of d must be at least $\max(1, n)$. e contains the off-diagonal elements of T . The dimension of e must be at least $\max(1, n-1)$.

Output Parameters

d	The n eigenvalues in ascending order, unless <code>info > 0</code> . See also <code>info</code> .
e	On exit, the array is overwritten; see <code>info</code> .

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = i`, the algorithm failed to find all the eigenvalues after $30n$ iterations:

i off-diagonal elements have not converged to zero. On exit, d and e contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to T .

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If λ_i is an exact eigenvalue, and m_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \epsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about $14n^2$.

?steqr

Computes all eigenvalues and eigenvectors of a symmetric or Hermitian matrix reduced to tridiagonal form (QR algorithm).

Syntax

```
lapack_int LAPACKE_ssteqr( int matrix_layout, char compz, lapack_int n, float* d,
float* e, float* z, lapack_int ldz );

lapack_int LAPACKE_dsteqr( int matrix_layout, char compz, lapack_int n, double* d,
double* e, double* z, lapack_int ldz );

lapack_int LAPACKE_csteqr( int matrix_layout, char compz, lapack_int n, float* d,
float* e, lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zsteqr( int matrix_layout, char compz, lapack_int n, double* d,
double* e, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric tridiagonal matrix T . In other words, the routine can compute the spectral factorization: $T = Z * \Lambda * Z^T$. Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i ; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$$T * z_i = \lambda_i * z_i \text{ for } i = 1, 2, \dots, n.$$

The routine normalizes the eigenvectors so that $\|z_i\|_2 = 1$.

You can also use the routine for computing the eigenvalues and eigenvectors of an arbitrary real symmetric (or complex Hermitian) matrix A reduced to tridiagonal form T : $A = Q * T * Q^H$. In this case, the spectral factorization is as follows: $A = Q * T * Q^H = (Q * Z) * \Lambda * (Q * Z)^H$. Before calling ?steqr, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	?sytrd, ?orgtr	?hetrd, ?ungtr
packed storage	?sptrd, ?opgtr	?hptrd, ?upgtr

	for real matrices:	for complex matrices:
band storage	?sbtrd(<i>vect</i> ='V')	?hbtrd(<i>vect</i> ='V')

If you need eigenvalues only, it's more efficient to call [sterf](#). If T is positive-definite, [pteqr](#) can compute small eigenvalues more accurately than [steqr](#).

To solve the problem by a single call, use one of the divide and conquer routines [stevd](#), [syevd](#), [spevd](#), or [sbevd](#) for real symmetric matrices or [heevd](#), [hpevd](#), or [hbevd](#) for complex Hermitian matrices.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>compz</i>	Must be 'N' or 'I' or 'V'. If <i>compz</i> = 'N', the routine computes eigenvalues only. If <i>compz</i> = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T . If <i>compz</i> = 'V', the routine computes the eigenvalues and eigenvectors of the original symmetric matrix. On entry, z must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
<i>n</i>	The order of the matrix T ($n \geq 0$).
<i>d</i> , <i>e</i>	Arrays: d contains the diagonal elements of T . The size of d must be at least $\max(1, n)$. e contains the off-diagonal elements of T . The size of e must be at least $\max(1, n-1)$.
<i>z</i>	Array, size $\max(1, ldz*n)$. If <i>compz</i> = 'N' or 'I', z need not be set. If <i>vect</i> = 'V', z must contain the orthogonal matrix used in the reduction to tridiagonal form.
<i>ldz</i>	The leading dimension of z . Constraints: $ldz \geq 1$ if <i>compz</i> = 'N'; $ldz \geq \max(1, n)$ if <i>compz</i> = 'V' or 'I'.

Output Parameters

<i>d</i>	The n eigenvalues in ascending order, unless <i>info</i> > 0. See also <i>info</i> .
<i>e</i>	On exit, the array is overwritten; see <i>info</i> .
<i>z</i>	If <i>info</i> = 0, contains the n -by- n matrix the columns of which are orthonormal eigenvectors (the i -th column corresponds to the i -th eigenvalue).

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = *i*, the algorithm failed to find all the eigenvalues after $30n$ iterations: *i* off-diagonal elements have not converged to zero. On exit, *d* and *e* contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to *T*.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\varepsilon) * \|T\|_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \varepsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n) * \varepsilon * \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

$24n^2$ if *compz* = 'N';

$7n^3$ (for complex flavors, $14n^3$) if *compz* = 'V' or 'I'.

?stemr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
lapack_int LAPACKESstemr( int matrix_layout, char jobz, char range, lapack_int n,
const float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, lapack_int*
m, float* w, float* z, lapack_int ldz, lapack_int nzc, lapack_int* isuppz,
lapack_logical* tryrac );

lapack_int LAPACKEdstemr( int matrix_layout, char jobz, char range, lapack_int n,
const double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu,
lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int nzc, lapack_int* isuppz,
lapack_logical* tryrac );

lapack_int LAPACKEcstemr( int matrix_layout, char jobz, char range, lapack_int n,
const float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, lapack_int*
m, float* w, lapack_complex_float* z, lapack_int ldz, lapack_int nzc, lapack_int*
isuppz, lapack_logical* tryrac );

lapack_int LAPACKEZstemr( int matrix_layout, char jobz, char range, lapack_int n,
const double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu,
lapack_int* m, double* w, lapack_complex_double* z, lapack_int ldz, lapack_int nzc,
lapack_int* isuppz, lapack_logical* tryrac );
```

Include Files

- `mkl.h`

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . Any such unreduced matrix has a well defined set of pairwise different real eigenvalues, the corresponding real eigenvectors are pairwise orthogonal.

The spectrum may be computed either completely or partially by specifying either an interval $(vl, vu]$ or a range of indices $il:iu$ for the desired eigenvalues.

Depending on the number of desired eigenvalues, these are computed either by bisection or the *dqds* algorithm. Numerically orthogonal eigenvectors are computed by the use of various suitable L^*D*L^T factorizations near clusters of close eigenvalues (referred to as RRRs, Relatively Robust Representations). An informal sketch of the algorithm follows.

For each unreduced block (submatrix) of T ,

- a.** Compute $T - \sigma I = L^*D^*L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of L and D cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- b.** Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see steps c and d.
- c.** For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- d.** For each eigenvalue with a large enough relative separation compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to step c for any clusters that remain.

Normal execution of `?stemr` may create NaNs and infinities and may abort due to a floating point exception in environments that do not handle NaNs and infinities in the IEEE standard default manner.

For more details, see: [Dhillon04], [Dhillon04-02], [Dhillon97]

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobz</code>	Must be 'N' or 'V'. If <code>jobz = 'N'</code> , then only eigenvalues are computed. If <code>jobz = 'V'</code> , then eigenvalues and eigenvectors are computed.
<code>range</code>	Must be 'A' or 'V' or 'I'. If <code>range = 'A'</code> , the routine computes all eigenvalues. If <code>range = 'V'</code> , the routine computes all eigenvalues in the half-open interval: $(vl, vu]$. If <code>range = 'I'</code> , the routine computes eigenvalues with indices il to iu .
<code>n</code>	The order of the matrix T ($n \geq 0$).
<code>d</code>	Array, size n .

	Contains n diagonal elements of the tridiagonal matrix T .
e	<p>Array, size n.</p> <p>Contains $(n-1)$ off-diagonal elements of the tridiagonal matrix T in elements 0 to $n-2$ of e. $e[n-1]$ need not be set on input, but is used internally as workspace.</p>
vl, vu	<p>If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$.</p> <p>If $range = 'A'$ or $'I'$, vl and vu are not referenced.</p>
il, iu	<p>If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$.</p> <p>If $range = 'A'$ or $'V'$, il and iu are not referenced.</p>
ldz	<p>The leading dimension of the output array z.</p> <p>if $jobz = 'V'$, then $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout ;</p> <p>$ldz \geq 1$ otherwise.</p>
nzc	<p>The number of eigenvectors to be held in the array z.</p> <p>If $range = 'A'$, then $nzc \geq \max(1, n)$;</p> <p>If $range = 'V'$, then nzc is greater than or equal to the number of eigenvalues in the half-open interval: $(vl, vu]$.</p> <p>If $range = 'I'$, then $nzc \geq iu - il + 1$.</p> <p>If $nzc = -1$, then a workspace query is assumed; the routine calculates the number of columns of the array z that are needed to hold the eigenvectors.</p> <p>This value is returned as the first entry of the array z, and no error message related to nzc is issued by the routine <code>xerbla</code>.</p>
$tryrac$	<p>If $tryrac$ is true, it indicates that the code should check whether the tridiagonal matrix defines its eigenvalues to high relative accuracy. If so, the code uses relative-accuracy preserving algorithms that might be (a bit) slower depending on the matrix. If the matrix does not define its eigenvalues to high relative accuracy, the code can use possibly faster algorithms.</p> <p>If $tryrac$ is not true, the code is not required to guarantee relatively accurate eigenvalues and can use the fastest possible techniques.</p>

Output Parameters

d	On exit, the array d is overwritten.
e	On exit, the array e is overwritten.
m	<p>The total number of eigenvalues found, $0 \leq m \leq n$.</p> <p>If $range = 'A'$, then $m=n$, and if $range = 'I'$, then $m=iu-il+1$.</p>

<i>w</i>	<p>Array, size <i>n</i>.</p> <p>The first <i>m</i> elements contain the selected eigenvalues in ascending order.</p>
<i>z</i>	<p>Array <i>z</i>(size max(1, <i>ldz</i>*<i>m</i>) for column major layout and max(1, <i>ldz</i>*<i>n</i>) for row major layout) .</p> <p>If <i>jobz</i> = 'V', and <i>info</i> = 0, then the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>T</i> corresponding to the selected eigenvalues, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with <i>w</i>(<i>i</i>) .</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p> <p>Note: the exact value of <i>m</i> is not known in advance and can be computed with a workspace query by setting <i>nzc</i>=-1, see description of the parameter <i>nzc</i>.</p>
<i>isuppz</i>	<p>Array, size (2*max(1, <i>m</i>)).</p> <p>The support of the eigenvectors in <i>z</i>, that is the indices indicating the nonzero elements in <i>z</i>. The <i>i</i>-th computed eigenvector is nonzero only in elements <i>isuppz</i>[2*<i>i</i> - 2] through <i>isuppz</i>[2*<i>i</i> - 1]. This is relevant in the case when the matrix is split. <i>isuppz</i> is only accessed when <i>jobz</i> = 'V' and <i>n</i>>0.</p>
<i>tryrac</i>	<p>On exit, , set to true. <i>tryrac</i> is set to false if the matrix does not define its eigenvalues to high relative accuracy.</p>

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, an internal error occurred.

?stedc

Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Syntax

```
lapack_int LAPACKE_sstedc( int matrix_layout, char compz, lapack_int n, float* d,
float* e, float* z, lapack_int ldz );

lapack_int LAPACKE_dstedc( int matrix_layout, char compz, lapack_int n, double* d,
double* e, double* z, lapack_int ldz );

lapack_int LAPACKE_cstedc( int matrix_layout, char compz, lapack_int n, float* d,
float* e, lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zstedc( int matrix_layout, char compz, lapack_int n, double* d,
double* e, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band real symmetric or complex Hermitian matrix can also be found if [sytrd/hetrd](#) or [sptrd/hptrd](#) or [sbtrd/hbtrd](#) has been used to reduce this matrix to tridiagonal form.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>compz</i>	<p>Must be 'N' or 'I' or 'V'.</p> <p>If <i>compz</i> = 'N', the routine computes eigenvalues only.</p> <p>If <i>compz</i> = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix.</p> <p>If <i>compz</i> = 'V', the routine computes the eigenvalues and eigenvectors of original symmetric/Hermitian matrix. On entry, the array <i>z</i> must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.</p>
<i>n</i>	The order of the symmetric tridiagonal matrix ($n \geq 0$).
<i>d</i> , <i>e</i>	<p>Arrays:</p> <p><i>d</i> contains the diagonal elements of the tridiagonal matrix.</p> <p>The dimension of <i>d</i> must be at least $\max(1, n)$.</p> <p><i>e</i> contains the subdiagonal elements of the tridiagonal matrix.</p> <p>The dimension of <i>e</i> must be at least $\max(1, n-1)$.</p>
<i>z</i>	<p>Array <i>z</i> is of size $\max(1, ldz * n)$.</p> <p>If <i>compz</i> = 'V', then, on entry, <i>z</i> must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.</p>
<i>ldz</i>	<p>The leading dimension of <i>z</i>. Constraints:</p> <p>$ldz \geq 1$ if <i>compz</i> = 'N';</p> <p>$ldz \geq \max(1, n)$ if <i>compz</i> = 'V' or 'I'.</p>

Output Parameters

<i>d</i>	<p>The <i>n</i> eigenvalues in ascending order, unless <i>info</i> $\neq 0$.</p> <p>See also <i>info</i>.</p>
<i>e</i>	On exit, the array is overwritten; see <i>info</i> .
<i>z</i>	<p>If <i>info</i> = 0, then if <i>compz</i> = 'V', <i>z</i> contains the orthonormal eigenvectors of the original symmetric/Hermitian matrix, and if <i>compz</i> = 'I', <i>z</i> contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If <i>compz</i> = 'N', <i>z</i> is not referenced.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *i*/(*n*+1) through mod(*i*, *n*+1).

?stegr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
lapack_int LAPACKE_sstegr( int matrix_layout, char jobz, char range, lapack_int n,
float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* isuppz );

lapack_int LAPACKE_dstegr( int matrix_layout, char jobz, char range, lapack_int n,
double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu, double abstol,
lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* isuppz );

lapack_int LAPACKE_cstegr( int matrix_layout, char jobz, char range, lapack_int n,
float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz, lapack_int* isuppz );

lapack_int LAPACKE_zstegr( int matrix_layout, char jobz, char range, lapack_int n,
double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu, double abstol,
lapack_int* m, double* w, lapack_complex_double* z, lapack_int ldz, lapack_int*
isuppz );
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *T*.

The spectrum may be computed either completely or partially by specifying either an interval (*vl*, *vu*] or a range of indices *il*:*iu* for the desired eigenvalues.

?stegr is a compatibility wrapper around the improved [stemr](#) routine. See its description for further details.

Note that the *abstol* parameter no longer provides any benefit and hence is no longer used.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.

<i>range</i>	<p>Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval:</p> $vl < w[i] \leq vu.$ <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p>
<i>n</i>	The order of the matrix T ($n \geq 0$).
<i>d</i> , <i>e</i>	<p>Arrays:</p> <p><i>d</i> contains the diagonal elements of T.</p> <p>The dimension of <i>d</i> must be at least $\max(1, n)$.</p> <p><i>e</i> contains the subdiagonal elements of T in elements 1 to $n-1$; $e(n)$ need not be set on input, but it is used as a workspace.</p> <p>The dimension of <i>e</i> must be at least $\max(1, n)$.</p>
<i>vl</i> , <i>vu</i>	<p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $vl < vu$.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il</i> , <i>iu</i>	<p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$.</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	Unused. Was the absolute error tolerance for the eigenvalues/eigenvectors in previous versions.
<i>ldz</i>	<p>The leading dimension of the output array <i>z</i>. Constraints:</p> $ldz \geq 1 \text{ if } jobz = 'N';$ $ldz \geq \max(1, n) \text{ if } jobz = 'V'.$

Output Parameters

<i>d</i> , <i>e</i>	On exit, <i>d</i> and <i>e</i> are overwritten.
<i>m</i>	<p>The total number of eigenvalues found,</p> $0 \leq m \leq n.$ <p>If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.</p>
<i>w</i>	<p>Array, size at least $\max(1, n)$.</p> <p>The selected eigenvalues in ascending order, stored in $w[0]$ to $w[m - 1]$.</p>
<i>z</i>	Array $z(\text{size } \max(1, ldz * m))$.

If `jobz = 'V'`, and if `info = 0`, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w[i - 1]$.

If `jobz = 'N'`, then z is not referenced.

Note: if `range = 'V'`, the exact value of m is not known in advance and an upper bound must be used. Using $n = m$ is always safe.

`isuppz`

Array, size at least $(2 * \max(1, m))$.

The support of the eigenvectors in z , that is the indices indicating the nonzero elements in z . The i -th computed eigenvector is nonzero only in elements `isuppz[2*i - 2]` through `isuppz[2*i - 1]`. This is relevant in the case when the matrix is split. `isuppz` is only accessed when `jobz = 'V'`, and $n > 0$.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info > 0`, an internal error occurred.

?pteqr

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric positive-definite tridiagonal matrix.

Syntax

```
lapack_int LAPACKE_spteqr( int matrix_layout, char compz, lapack_int n, float* d,
float* e, float* z, lapack_int ldz );
```

```
lapack_int LAPACKE_dpteqr( int matrix_layout, char compz, lapack_int n, double* d,
double* e, double* z, lapack_int ldz );
```

```
lapack_int LAPACKE_cpteqr( int matrix_layout, char compz, lapack_int n, float* d,
float* e, lapack_complex_float* z, lapack_int ldz );
```

```
lapack_int LAPACKE_zpteqr( int matrix_layout, char compz, lapack_int n, double* d,
double* e, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- `mkl.h`

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric positive-definite tridiagonal matrix T . In other words, the routine can compute the spectral factorization: $T = Z * \Lambda * Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i ; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$$T^* z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

(The routine normalizes the eigenvectors so that $\|z_i\|_2 = 1$.)

You can also use the routine for computing the eigenvalues and eigenvectors of real symmetric (or complex Hermitian) positive-definite matrices A reduced to tridiagonal form T : $A = Q^* T Q^H$. In this case, the spectral factorization is as follows: $A = Q^* T^* Q^H = (QZ)^* \Lambda (QZ)^H$. Before calling `?pteqr`, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	<code>?sytrd, ?orgtr</code>	<code>?hetrd, ?ungtr</code>
packed storage	<code>?sptrd, ?opgtr</code>	<code>?hptrd, ?upgtr</code>
band storage	<code>?sbtrd(vect='V')</code>	<code>?hbtrd(vect='V')</code>

The routine first factorizes T as $L^* D L^H$ where L is a unit lower bidiagonal matrix, and D is a diagonal matrix. Then it forms the bidiagonal matrix $B = L^* D^{1/2}$ and calls `?bdsqr` to compute the singular values of B , which are the square roots of the eigenvalues of T .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>compz</code>	Must be 'N' or 'I' or 'V'. If <code>compz = 'N'</code> , the routine computes eigenvalues only. If <code>compz = 'I'</code> , the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T . If <code>compz = 'V'</code> , the routine computes the eigenvalues and eigenvectors of A (and the array z must contain the matrix Q on entry).
<code>n</code>	The order of the matrix T ($n \geq 0$).
<code>d, e</code>	Arrays: d contains the diagonal elements of T . The size of d must be at least $\max(1, n)$. e contains the off-diagonal elements of T . The size of e must be at least $\max(1, n-1)$.
<code>z</code>	Array, size $\max(1, ldz * n)$ If <code>compz = 'N' or 'I'</code> , z need not be set. If <code>compz = 'V'</code> , z must contain the orthogonal matrix used in the reduction to tridiagonal form..
<code>ldz</code>	The leading dimension of z . Constraints: $ldz \geq 1$ if <code>compz = 'N'</code> ; $ldz \geq \max(1, n)$ if <code>compz = 'V' or 'I'</code> .

Output Parameters

<i>d</i>	The n eigenvalues in descending order, unless <i>info</i> > 0. See also <i>info</i> .
<i>e</i>	On exit, the array is overwritten.
<i>z</i>	If <i>info</i> = 0, contains an n -by- n matrix the columns of which are orthonormal eigenvectors. (The i -th column corresponds to the i -th eigenvalue.)

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = i , the leading minor of order i (and hence T itself) is not positive-definite.

If *info* = $n + i$, the algorithm for computing singular values failed to converge; i off-diagonal elements have not converged to zero.

If *info* = $-i$, the i -th parameter had an illegal value.

Application Notes

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \varepsilon * K * \lambda_i$$

where $c(n)$ is a modestly increasing function of n , ε is the machine precision, and $K = ||DTD||_2 * ||(DTD)^{-1}||_2$, D is diagonal with $d_{ii} = t_{ii}^{-1/2}$.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(u_i, w_i) \leq c(n) \varepsilon K / \min_{i \neq j} (|\lambda_i - \lambda_j| / |\lambda_i + \lambda_j|).$$

Here $\min_{i \neq j} (|\lambda_i - \lambda_j| / |\lambda_i + \lambda_j|)$ is the *relative gap* between λ_i and the other eigenvalues.

The total number of floating-point operations depends on how rapidly the algorithm converges.

Typically, it is about

$30n^2$ if *compz* = 'N';

$6n^3$ (for complex flavors, $12n^3$) if *compz* = 'V' or 'I'.

?stebz

Computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.

Syntax

```
lapack_int LAPACKE_sstebz (char range, char order, lapack_int n, float vl, float vu,
lapack_int il, lapack_int iu, float abstol, const float* d, const float* e, lapack_int*
m, lapack_int* nsplit, float* w, lapack_int* iblock, lapack_int* isplit);
```

```
lapack_int LAPACKE_dstebz (char range, char order, lapack_int n, double vl, double vu,
lapack_int il, lapack_int iu, double abstol, const double* d, const double* e,
lapack_int* m, lapack_int* nsplit, double* w, lapack_int* iblock, lapack_int* isplit);
```

Include Files

- `mkl.h`

Description

The routine computes some (or all) of the eigenvalues of a real symmetric tridiagonal matrix T by bisection. The routine searches for zero or negligible off-diagonal elements to see if T splits into block-diagonal form $T = \text{diag}(T_1, T_2, \dots)$. Then it performs bisection on each of the blocks T_i and returns the block index of each computed eigenvalue, so that a subsequent call to [stein](#) can also take advantage of the block structure.

Input Parameters

<i>range</i>	<p>Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$.</p> <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p>
<i>order</i>	<p>Must be 'B' or 'E'.</p> <p>If <i>order</i> = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block.</p> <p>If <i>order</i> = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.</p>
<i>n</i>	The order of the matrix T ($n \geq 0$).
<i>vl, vu</i>	<p>If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval:</p> $vl < w[i] \leq vu.$ <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il, iu</i>	<p>Constraint: $1 \leq il \leq iu \leq n$.</p> <p>If <i>range</i> = 'I', the routine computes eigenvalues $w[i]$ such that $il \leq i \leq iu$ (assuming that the eigenvalues $w[i]$ are in ascending order).</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	<p>The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width <i>abstol</i>.</p> <p>If $abstol \leq 0.0$, then the tolerance is taken as $\text{eps} * T$, where <i>eps</i> is the machine precision, and T is the 1-norm of the matrix T.</p>
<i>d, e</i>	<p>Arrays:</p> <p><i>d</i> contains the diagonal elements of T.</p> <p>The size of <i>d</i> must be at least $\max(1, n)$.</p> <p><i>e</i> contains the off-diagonal elements of T.</p> <p>The size of <i>e</i> must be at least $\max(1, n-1)$.</p>

Output Parameters

<i>m</i>	The actual number of eigenvalues found.
<i>nsplit</i>	The number of diagonal blocks detected in <i>T</i> .
<i>w</i>	Array, size at least $\max(1, n)$. The computed eigenvalues, stored in <i>w</i> [0] to <i>w</i> [<i>m</i> - 1].
<i>iblock, isplit</i>	<p>Arrays, size at least $\max(1, n)$.</p> <p>A positive value <i>iblock</i>[<i>i</i>] is the block number of the eigenvalue stored in <i>w</i>[<i>i</i>] (see also <i>info</i>).</p> <p>The leading <i>nsplit</i> elements of <i>isplit</i> contain points at which <i>T</i> splits into blocks <i>T_i</i> as follows: the block <i>T₁</i> contains rows/columns 1 to <i>isplit</i>[0]; the block <i>T₂</i> contains rows/columns <i>isplit</i>[0]+1 to <i>isplit</i>[1], and so on.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = 1, for *range* = 'A' or 'V', the algorithm failed to compute some of the required eigenvalues to the desired accuracy; *iblock*[*i*] < 0 indicates that the eigenvalue stored in *w*[*i*] failed to converge.

If *info* = 2, for *range* = 'I', the algorithm failed to compute some of the required eigenvalues. Try calling the routine again with *range* = 'A'.

If *info* = 3:

for *range* = 'A' or 'V', same as *info* = 1;

for *range* = 'I', same as *info* = 2.

If *info* = 4, no eigenvalues have been computed. The floating-point arithmetic on the computer is not behaving as expected.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The eigenvalues of *T* are computed to high relative accuracy which means that if they vary widely in magnitude, then any small eigenvalues will be computed more accurately than, for example, with the standard QR method. However, the reduction to tridiagonal form (prior to calling the routine) may exclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix if its eigenvalues vary widely in magnitude.

?stein

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix.

Syntax

```
lapack_int LAPACKE_sstein( int matrix_layout, lapack_int n, const float* d, const
float* e, lapack_int m, const float* w, const lapack_int* iblock, const lapack_int*
isplit, float* z, lapack_int ldz, lapack_int* ifailv );
```

```
lapack_int LAPACKE_dstein( int matrix_layout, lapack_int n, const double* d, const
double* e, lapack_int m, const double* w, const lapack_int* iblock, const lapack_int*
isplit, double* z, lapack_int ldz, lapack_int* ifailv );
```

```
lapack_int LAPACKE_cstein( int matrix_layout, lapack_int n, const float* d, const
float* e, lapack_int m, const float* w, const lapack_int* iblock, const lapack_int*
isplit, lapack_complex_float* z, lapack_int ldz, lapack_int* ifailv );
```

```
lapack_int LAPACKE_zstein( int matrix_layout, lapack_int n, const double* d, const
double* e, lapack_int m, const double* w, const lapack_int* iblock, const lapack_int*
isplit, lapack_complex_double* z, lapack_int ldz, lapack_int* ifailv );
```

Include Files

- `mkl.h`

Description

The routine computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, by inverse iteration. It is designed to be used in particular after the specified eigenvalues have been computed by `?stebz` with `order = 'B'`, but may also be used when the eigenvalues have been computed by other routines.

If you use this routine after `?stebz`, it can take advantage of the block structure by performing inverse iteration on each block T_i separately, which is more efficient than using the whole matrix T .

If T has been formed by reduction of a full symmetric or Hermitian matrix A to tridiagonal form, you can transform eigenvectors of T to eigenvectors of A by calling `?ormtr` or `?opmtr` (for real flavors) or by calling `?unmtr` or `?upmtr` (for complex flavors).

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>n</code>	The order of the matrix T ($n \geq 0$).
<code>m</code>	The number of eigenvectors to be returned.
<code>d, e, w</code>	Arrays: d contains the diagonal elements of T . The size of d must be at least $\max(1, n)$. e contains the sub-diagonal elements of T stored in elements 1 to $n-1$. The size of e must be at least $\max(1, n-1)$. w contains the eigenvalues of T , stored in $w[0]$ to $w[m-1]$ (as returned by <code>stebz</code>). Eigenvalues of T_1 must be supplied first, in non-decreasing order; then those of T_2 , again in non-decreasing order, and so on. Constraint: if <code>iblock[i] = iblock[i+1]</code> , <code>w[i] ≤ w[i+1]</code> . The size of w must be at least $\max(1, n)$.
<code>iblock, isplit</code>	Arrays, size at least $\max(1, n)$. The arrays <code>iblock</code> and <code>isplit</code> , as returned by <code>?stebz</code> with <code>order = 'B'</code> .

If you did not call `?stebz` with `order = 'B'`, set all elements of `iblock` to 1, and `isplit[0]` to `n`.)

`ldz`

The leading dimension of the output array `z`; $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout.

Output Parameters

`z`

Array, size at least $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout.

If `info = 0`, `z` contains an n -by- n matrix the columns of which are orthonormal eigenvectors. (The i -th column corresponds to the i th eigenvalue.)

`ifailv`

Array, size at least $\max(1, m)$.

If `info = i > 0`, the first i elements of `ifailv` contain the indices of any eigenvectors that failed to converge.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = i`, then i eigenvectors (as indicated by the parameter `ifailv`) each failed to converge in 5 iterations. The current iterates are stored in the corresponding columns/rows of the array `z`.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

Each computed eigenvector z_i is an exact eigenvector of a matrix $T + E_i$, where $\|E_i\|_2 = O(\epsilon) * \|T\|_2$. However, a set of eigenvectors computed by this routine may not be orthogonal to so high a degree of accuracy as those computed by `?steqr`.

`?disna`

Computes the reciprocal condition numbers for the eigenvectors of a symmetric/ Hermitian matrix or for the left or right singular vectors of a general matrix.

Syntax

```
lapack_int LAPACKE_sdisna (char job, lapack_int m, lapack_int n, const float* d, float* sep);
```

```
lapack_int LAPACKE_ddisna (char job, lapack_int m, lapack_int n, const double* d, double* sep);
```

Include Files

- `mk1.h`

Description

The routine computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general m -by- n matrix.

The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the i -th computed vector is given by

`?lamch('E')*(anorm/sep(i))`

where $anorm = ||A||_2 = \max(|d(j)|)$. $sep(i)$ is not allowed to be smaller than `slamch('E')*anorm` in order to limit the size of the error bound.

`?disna` may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

Input Parameters

<i>job</i>	Must be 'E', 'L', or 'R'. Specifies for which problem the reciprocal condition numbers should be computed: <i>job</i> = 'E': for the eigenvectors of a symmetric/Hermitian matrix; <i>job</i> = 'L': for the left singular vectors of a general matrix; <i>job</i> = 'R': for the right singular vectors of a general matrix.
<i>m</i>	The number of rows of the matrix ($m \geq 0$).
<i>n</i>	If <i>job</i> = 'L', or 'R', the number of columns of the matrix ($n \geq 0$). Ignored if <i>job</i> = 'E'.
<i>d</i>	Array, dimension at least $\max(1, m)$ if <i>job</i> = 'E', and at least $\max(1, \min(m, n))$ if <i>job</i> = 'L' or 'R'. This array must contain the eigenvalues (if <i>job</i> = 'E') or singular values (if <i>job</i> = 'L' or 'R') of the matrix, in either increasing or decreasing order. If singular values, they must be non-negative.

Output Parameters

<i>sep</i>	Array, dimension at least $\max(1, m)$ if <i>job</i> = 'E', and at least $\max(1, \min(m, n))$ if <i>job</i> = 'L' or 'R'. The reciprocal condition numbers of the vectors.
------------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Generalized Symmetric-Definite Eigenvalue Problems: LAPACK Computational Routines

Generalized symmetric-definite eigenvalue problems are as follows: find the eigenvalues λ and the corresponding eigenvectors z that satisfy one of these equations:

$$Az = \lambda Bz, ABz = \lambda z, \text{ or } BAz = \lambda z,$$

where A is an n -by- n symmetric or Hermitian matrix, and B is an n -by- n symmetric positive-definite or Hermitian positive-definite matrix.

In these problems, there exist n real eigenvectors corresponding to real eigenvalues (even for complex Hermitian matrices A and B).

Routines described in this topic allow you to reduce the above generalized problems to standard symmetric eigenvalue problem $CY = \lambda Y$, which you can solve by calling LAPACK routines described earlier in this chapter (see [Symmetric Eigenvalue Problems](#)).

Different routines allow the matrices to be stored either conventionally or in packed storage. Prior to reduction, the positive-definite matrix B must first be factorized using either [potrf](#) or [pptrf](#).

The reduction routine for the banded matrices A and B uses a split Cholesky factorization for which a specific routine [pbstf](#) is provided. This refinement halves the amount of work required to form matrix C .

Table "Computational Routines for Reducing Generalized Eigenproblems to Standard Problems" lists LAPACK routines that can be used to solve generalized symmetric-definite eigenvalue problems.

Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Matrix type	Reduce to standard problems (full storage)	Reduce to standard problems (packed storage)	Reduce to standard problems (band matrices)	Factorize band matrix
real symmetric matrices	sygst	spgst	sbgst	pbstf
complex Hermitian matrices	hegst	hpgst	hbgst	pbstf

?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

```
lapack_int LAPACKE_ssygst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, float* a, lapack_int lda, const float* b, lapack_int ldb);
```

```
lapack_int LAPACKE_dsygst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, double* a, lapack_int lda, const double* b, lapack_int ldb);
```

Include Files

- mkl.h

Description

The routine reduces real symmetric-definite generalized eigenproblems

$$A^*z = \lambda^*B^*z, \quad A^*B^*z = \lambda^*z, \quad \text{or} \quad B^*A^*z = \lambda^*z$$

to the standard form $C^*y = \lambda^*y$. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call [?potrf](#) to compute the Cholesky factorization: $B = U^T*U$ or $B = L*L^T$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>itype</i>	Must be 1 or 2 or 3.

If $itype = 1$, the generalized eigenproblem is $A*z = \lambda*B*z$

for $uplo = 'U'$: $C = \text{inv}(U^T)*A*\text{inv}(U)$, $z = \text{inv}(U)*y$;

for $uplo = 'L'$: $C = \text{inv}(L)*A*\text{inv}(L^T)$, $z = \text{inv}(L^T)*y$.

If $itype = 2$, the generalized eigenproblem is $A*B*z = \lambda*z$

for $uplo = 'U'$: $C = U*A*U^T$, $z = \text{inv}(U)*y$;

for $uplo = 'L'$: $C = L^T*A*L$, $z = \text{inv}(L^T)*y$.

If $itype = 3$, the generalized eigenproblem is $B*A*z = \lambda*z$

for $uplo = 'U'$: $C = U*A*U^T$, $z = U^T*y$;

for $uplo = 'L'$: $C = L^T*A*L$, $z = L*y$.

Must be 'U' or 'L'.

If $uplo = 'U'$, the array a stores the upper triangle of A ; you must supply B in the factored form $B = U^T*U$.

If $uplo = 'L'$, the array a stores the lower triangle of A ; you must supply B in the factored form $B = L*L^T$.

The order of the matrices A and B ($n \geq 0$).

Arrays:

a (size $\max(1, lda*n)$) contains the upper or lower triangle of A .

b (size $\max(1, ldb*n)$) contains the Cholesky-factored matrix B :

$B = U^T*U$ or $B = L*L^T$ (as returned by `?potrf`).

The leading dimension of a ; at least $\max(1, n)$.

The leading dimension of b ; at least $\max(1, n)$.

$uplo$

n

a, b

lda

ldb

Output Parameters

a

The upper or lower triangle of A is overwritten by the upper or lower triangle of C , as specified by the arguments $itype$ and $uplo$.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if $itype = 1$) or B (if $itype = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?hegst

Reduces a complex Hermitian positive-definite generalized eigenvalue problem to the standard form.

Syntax

```
lapack_int LAPACKE_chegst (int matrix_layout, lapack_int itype, char uplo, lapack_int
n, lapack_complex_float* a, lapack_int lda, const lapack_complex_float* b, lapack_int
ldb);
```

```
lapack_int LAPACKE_zhegst (int matrix_layout, lapack_int itype, char uplo, lapack_int
n, lapack_complex_double* a, lapack_int lda, const lapack_complex_double* b, lapack_int
ldb);
```

Include Files

- mkl.h

Description

The routine reduces a complex Hermitian positive-definite generalized eigenvalue problem to standard form.

<i>itype</i>	Problem	Result
1	$A^*x = \lambda^*B^*x$	A overwritten by $\text{inv}(U^H) * A * \text{inv}(U)$ or $\text{inv}(L) * A * \text{inv}(L^H)$
2	$A^*B^*x = \lambda^*x$	A overwritten by $U^*A^*U^H$ or L^H*A*L
3	$B^*A^*x = \lambda^*x$	

Before calling this routine, you must call ?potrf to compute the Cholesky factorization: $B = U^H * U$ or $B = L^*L^H$.

Input Parameters

itype

Must be 1 or 2 or 3.

If *itype* = 1, the generalized eigenproblem is $A^*z = \text{lambda}^*B^*z$

for *uplo* = 'U': $C = (U^H)^{-1} * A^* U^{-1}$;

for *uplo* = 'L': $C = L^{-1} * A^* (L^H)^{-1}$.

If *itype* = 2, the generalized eigenproblem is $A^*B^*z = \text{lambda}^*z$

for *uplo* = 'U': $C = U^*A^*U^H$;

for *uplo* = 'L': $C = L^H*A*L$.

If *itype* = 3, the generalized eigenproblem is $B^*A^*z = \text{lambda}^*z$

for *uplo* = 'U': $C = U^*A^*U^H$;

for *uplo* = 'L': $C = L^H*A*L$.

uplo

Must be 'U' or 'L'.

If *uplo* = 'U', the array *a* stores the upper triangle of *A*; you must supply *B* in the factored form $B = U^H * U$.

If `uplo = 'L'`, the array `a` stores the lower triangle of `A`; you must supply `B` in the factored form $B = L * L^H$.

`n`

The order of the matrices `A` and `B` ($n \geq 0$).

`a, b`

Arrays:

`a` (size $\max(1, lda * n)$) contains the upper or lower triangle of `A`.

`b` (size $\max(1, ldb * n)$) contains the Cholesky-factored matrix `B`:

$B = U^H * U$ or $B = L * L^H$ (as returned by `?potrf`).

`lda`

The leading dimension of `a`; at least $\max(1, n)$.

`ldb`

The leading dimension of `b`; at least $\max(1, n)$.

Output Parameters

`a`

The upper or lower triangle of `A` is overwritten by the upper or lower triangle of `C`, as specified by the arguments `itype` and `uplo`.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

Application Notes

Forming the reduced matrix `C` is a stable procedure. However, it involves implicit multiplication by B^{-1} (if `itype = 1`) or `B` (if `itype = 2` or `3`). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if `B` is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?spgst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

```
lapack_int LAPACKE_sspgst (int matrix_layout, lapack_int itype, char uplo, lapack_int
n, float* ap, const float* bp);
```

```
lapack_int LAPACKE_dspgst (int matrix_layout, lapack_int itype, char uplo, lapack_int
n, double* ap, const double* bp);
```

Include Files

- `mkl.h`

Description

The routine reduces real symmetric-definite generalized eigenproblems

$A * x = \lambda * B * x$, $A * B * x = \lambda * x$, or $B * A * x = \lambda * x$

to the standard form $C*y = \lambda*y$, using packed matrix storage. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call `?pptrf` to compute the Cholesky factorization: $B = U^T*U$ or $B = L*L^T$.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>itype</code>	<p>Must be 1 or 2 or 3.</p> <p>If <code>itype = 1</code>, the generalized eigenproblem is $A*z = \lambda*B*z$ for <code>uplo = 'U'</code>: $C = \text{inv}(U^T)*A*\text{inv}(U)$, $z = \text{inv}(U)*y$; for <code>uplo = 'L'</code>: $C = \text{inv}(L)*A*\text{inv}(L^T)$, $z = \text{inv}(L^T)*y$.</p> <p>If <code>itype = 2</code>, the generalized eigenproblem is $A*B*z = \lambda*z$ for <code>uplo = 'U'</code>: $C = U*A*U^T$, $z = \text{inv}(U)*y$; for <code>uplo = 'L'</code>: $C = L^T*A*L$, $z = \text{inv}(L^T)*y$.</p> <p>If <code>itype = 3</code>, the generalized eigenproblem is $B*A*z = \lambda*z$ for <code>uplo = 'U'</code>: $C = U*A*U^T$, $z = U^T*y$; for <code>uplo = 'L'</code>: $C = L^T*A*L$, $z = L*y$.</p>
<code>uplo</code>	<p>Must be 'U' or 'L'.</p> <p>If <code>uplo = 'U'</code>, <code>ap</code> stores the packed upper triangle of A; you must supply B in the factored form $B = U^T*U$.</p> <p>If <code>uplo = 'L'</code>, <code>ap</code> stores the packed lower triangle of A; you must supply B in the factored form $B = L*L^T$.</p>
<code>n</code>	The order of the matrices A and B ($n \geq 0$).
<code>ap, bp</code>	<p>Arrays:</p> <p><code>ap</code> contains the packed upper or lower triangle of A. The dimension of <code>ap</code> must be at least $\max(1, n*(n+1)/2)$.</p> <p><code>bp</code> contains the packed Cholesky factor of B (as returned by <code>?pptrf</code> with the same <code>uplo</code> value). The dimension of <code>bp</code> must be at least $\max(1, n*(n+1)/2)$.</p>

Output Parameters

<code>ap</code>	The upper or lower triangle of A is overwritten by the upper or lower triangle of C , as specified by the arguments <code>itype</code> and <code>uplo</code> .
-----------------	--

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if $\text{itype} = 1$) or B (if $\text{itype} = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?hpgst

Reduces a generalized eigenvalue problem with a Hermitian matrix to a standard eigenvalue problem using packed storage.

Syntax

```
lapack_int LAPACKE_chpgst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, lapack_complex_float* ap, const lapack_complex_float* bp);
```

```
lapack_int LAPACKE_zhpgst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, lapack_complex_double* ap, const lapack_complex_double* bp);
```

Include Files

- mkl.h

Description

The routine reduces generalized eigenproblems with Hermitian matrices

$$A^*z = \lambda^*B^*z, A^*B^*z = \lambda^*z, \text{ or } B^*A^*z = \lambda^*z.$$

to standard eigenproblems $C^*y = \lambda^*y$, using packed matrix storage. Here A is a complex Hermitian matrix, and B is a complex Hermitian positive-definite matrix. Before calling this routine, you must call `?pptrf` to compute the Cholesky factorization: $B = U^H * U$ or $B = L * L^H$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>itype</i>	<p>Must be 1 or 2 or 3.</p> <p>If $\text{itype} = 1$, the generalized eigenproblem is $A^*z = \lambda^*B^*z$</p> <p>for $\text{uplo} = 'U'$: $C = \text{inv}(U^H) * A * \text{inv}(U)$, $z = \text{inv}(U) * y$;</p> <p>for $\text{uplo} = 'L'$: $C = \text{inv}(L) * A * \text{inv}(L^H)$, $z = \text{inv}(L^H) * y$.</p> <p>If $\text{itype} = 2$, the generalized eigenproblem is $A^*B^*z = \lambda^*z$</p> <p>for $\text{uplo} = 'U'$: $C = U^*A^*U^H$, $z = \text{inv}(U) * y$;</p> <p>for $\text{uplo} = 'L'$: $C = L^H * A^*L$, $z = \text{inv}(L^H) * y$.</p> <p>If $\text{itype} = 3$, the generalized eigenproblem is $B^*A^*z = \lambda^*z$</p> <p>for $\text{uplo} = 'U'$: $C = U^*A^*U^H$, $z = U^H * y$;</p> <p>for $\text{uplo} = 'L'$: $C = L^H * A^*L$, $z = L * y$.</p>

<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangle of <i>A</i>; you must supply <i>B</i> in the factored form $B = U^H * U$.</p> <p>If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangle of <i>A</i>; you must supply <i>B</i> in the factored form $B = L * L^H$.</p>
<i>n</i>	The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ap</i> , <i>bp</i>	<p>Arrays:</p> <p><i>ap</i> contains the packed upper or lower triangle of <i>A</i>.</p> <p>The dimension of <i>a</i> must be at least $\max(1, n*(n+1)/2)$.</p> <p><i>bp</i> contains the packed Cholesky factor of <i>B</i> (as returned by <code>?pptrf</code> with the same <i>uplo</i> value).</p> <p>The dimension of <i>b</i> must be at least $\max(1, n*(n+1)/2)$.</p>

Output Parameters

<i>ap</i>	The upper or lower triangle of <i>A</i> is overwritten by the upper or lower triangle of <i>C</i> , as specified by the arguments <i>itype</i> and <i>uplo</i> .
-----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?sbgst

Reduces a real symmetric-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.

Syntax

```
lapack_int LAPACKE_ssbgst (int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, float* ab, lapack_int ldab, const float* bb, lapack_int
ldbb, float* x, lapack_int ldx);
```

```
lapack_int LAPACKE_dsbgst (int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, double* ab, lapack_int ldab, const double* bb, lapack_int
ldbb, double* x, lapack_int ldx);
```

Include Files

- mkl.h

Description

To reduce the real symmetric-definite generalized eigenproblem $A^*z = \lambda^*B^*z$ to the standard form $C^*y = \lambda^*y$, where A , B and C are banded, this routine must be preceded by a call to [pbstf](#), which computes the split Cholesky factorization of the positive-definite matrix B : $B = S^T * S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites A with $C = X^T * A * X$, where $X = \text{inv}(S) * Q$ and Q is an orthogonal matrix chosen (implicitly) to preserve the bandwidth of A . The routine also has an option to allow the accumulation of X , and then, if z is an eigenvector of C , X^*z is an eigenvector of the original system.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>vect</i>	Must be 'N' or 'V'. If <i>vect</i> = 'N', then matrix X is not returned; If <i>vect</i> = 'V', then matrix X is returned.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of A .
<i>n</i>	The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	The number of super- or sub-diagonals in B ($ka \geq kb \geq 0$).
<i>ab, bb</i>	<i>ab</i> (size at least $\max(1, ldab * n)$ for column major layout and at least $\max(1, ldab * (ka + 1))$ for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by <i>uplo</i>) in band storage format. <i>bb</i> (size at least $\max(1, ldbb * n)$ for column major layout and at least $\max(1, ldbb * (kb + 1))$ for row major layout) is an array containing the banded split Cholesky factor of B as specified by <i>uplo</i> , <i>n</i> and <i>kb</i> and returned by pbstf / pbstf .
<i>ldab</i>	The leading dimension of the array <i>ab</i> ; must be at least $ka+1$ for column major layout and $\max(1, n)$ for row major layout.
<i>ldbb</i>	The leading dimension of the array <i>bb</i> ; must be at least $kb+1$ for column major layout and $\max(1, n)$ for row major layout.
<i>ldx</i>	The leading dimension of the output array x . Constraints: if <i>vect</i> = 'N', then $ldx \geq 1$; if <i>vect</i> = 'V', then $ldx \geq \max(1, n)$.

Output Parameters

<i>ab</i>	On exit, this array is overwritten by the upper or lower triangle of <i>C</i> as specified by <i>uplo</i> .
<i>x</i>	<p>Array.</p> <p>If <i>vect</i> = 'V', then <i>x</i> (size at least $\max(1, \text{ldx} * n)$) contains the <i>n</i>-by-<i>n</i> matrix $X = \text{inv}(S) * Q$.</p> <p>If <i>vect</i> = 'N', then <i>x</i> is not referenced.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by $\text{inv}(B)$. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

If *ka* and *kb* are much less than *n* then the total number of floating-point operations is approximately $6n^2 * kb$, when *vect* = 'N'. Additional $(3/2)n^3 * (kb/ka)$ operations are required when *vect* = 'V'.

?hbgst

Reduces a complex Hermitian positive-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.

Syntax

```
lapack_int LAPACKChbgst (int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab, const
lapack_complex_float* bb, lapack_int ldbb, lapack_complex_float* x, lapack_int ldx);

lapack_int LAPACKzhbgst (int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int ldab, const
lapack_complex_double* bb, lapack_int ldbb, lapack_complex_double* x, lapack_int ldx);
```

Include Files

- mkl.h

Description

To reduce the complex Hermitian positive-definite generalized eigenproblem $A * z = \lambda * B * z$ to the standard form $C * x = \lambda * y$, where *A*, *B* and *C* are banded, this routine must be preceded by a call to [pbstf/pbstf](#), which computes the split Cholesky factorization of the positive-definite matrix *B*: $B = S^H * S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites *A* with $C = X^H * A * X$, where $X = \text{inv}(S) * Q$, and *Q* is a unitary matrix chosen (implicitly) to preserve the bandwidth of *A*. The routine also has an option to allow the accumulation of *X*, and then, if *z* is an eigenvector of *C*, $X * z$ is an eigenvector of the original system.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>vect</i>	Must be 'N' or 'V'. If <i>vect</i> = 'N', then matrix <i>X</i> is not returned; If <i>vect</i> = 'V', then matrix <i>X</i> is returned.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ka</i>	The number of super- or sub-diagonals in <i>A</i> ($ka \geq 0$).
<i>kb</i>	The number of super- or sub-diagonals in <i>B</i> ($ka \geq kb \geq 0$).
<i>ab, bb</i>	<i>ab</i> (size at least $\max(1, ldab*n)$ for column major layout and at least $\max(1, ldab*(ka + 1))$ for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. <i>bb</i> (size at least $\max(1, ldbb*n)$ for column major layout and at least $\max(1, ldbb*(kb + 1))$ for row major layout) is an array containing the banded split Cholesky factor of <i>B</i> as specified by <i>uplo</i> , <i>n</i> and <i>kb</i> and returned by pbstf/pbstf .
<i>ldab</i>	The leading dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1 for column major layout and $\max(1, n)$ for row major layout.
<i>ldbb</i>	The leading dimension of the array <i>bb</i> ; must be at least <i>kb</i> +1 for column major layout and $\max(1, n)$ for row major layout.
<i>ldx</i>	The leading dimension of the output array <i>x</i> . Constraints: if <i>vect</i> = 'N', then $ldx \geq 1$; if <i>vect</i> = 'V', then $ldx \geq \max(1, n)$.

Output Parameters

<i>ab</i>	On exit, this array is overwritten by the upper or lower triangle of <i>C</i> as specified by <i>uplo</i> .
<i>x</i>	Array. If <i>vect</i> = 'V', then <i>x</i> (size at least $\max(1, ldx*n)$) contains the <i>n</i> -by- <i>n</i> matrix $X = \text{inv}(S) * Q$. If <i>vect</i> = 'N', then <i>x</i> is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

Forming the reduced matrix *C* involves implicit multiplication by $\text{inv}(B)$. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion. The total number of floating-point operations is approximately $20n^2*kb$, when *vect* = 'N'. Additional $5n^3*(kb/ka)$ operations are required when *vect* = 'V'. All these estimates assume that both *ka* and *kb* are much less than *n*.

?pbstf

Computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite banded matrix used in ?sbgst/?hbgst.

Syntax

```
lapack_int LAPACKE_spbstf (int matrix_layout, char uplo, lapack_int n, lapack_int kb,
float* bb, lapack_int ldbb);
```

```
lapack_int LAPACKE_dpbstf (int matrix_layout, char uplo, lapack_int n, lapack_int kb,
double* bb, lapack_int ldbb);
```

```
lapack_int LAPACKE_cpbstf (int matrix_layout, char uplo, lapack_int n, lapack_int kb,
lapack_complex_float* bb, lapack_int ldbb);
```

```
lapack_int LAPACKE_zpbstf (int matrix_layout, char uplo, lapack_int n, lapack_int kb,
lapack_complex_double* bb, lapack_int ldbb);
```

Include Files

- mkl.h

Description

The routine computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite band matrix *B*. It is to be used in conjunction with [sbgst/hbgst](#).

The factorization has the form $B = S^T * S$ (or $B = S^H * S$ for complex flavors), where *S* is a band matrix of the same bandwidth as *B* and the following structure: *S* is upper triangular in the first $(n+kb)/2$ rows and lower triangular in the remaining rows.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>bb</i> stores the upper triangular part of <i>B</i> . If <i>uplo</i> = 'L', <i>bb</i> stores the lower triangular part of <i>B</i> .

n	The order of the matrix B ($n \geq 0$).
kb	The number of super- or sub-diagonals in B ($kb \geq 0$).
bb	bb (size at least $\max(1, ldbb*n)$ for column major layout and at least $\max(1, ldbb*(kb + 1))$ for row major layout) is an array containing either upper or lower triangular part of the matrix B (as specified by <i>uplo</i>) in band storage format.
$ldbb$	The leading dimension of bb ; must be at least $kb+1$ for column major and at least $\max(1, n)$ for row major.

Output Parameters

bb	On exit, this array is overwritten by the elements of the split Cholesky factor S .
------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = i , then the factorization could not be completed, because the updated element b_{ii} would be the square root of a negative number; hence the matrix B is not positive-definite.

If *info* = $-i$, the i -th parameter had an illegal value.

Application Notes

The computed factor S is the exact factor of a perturbed matrix $B + E$, where

$$|E| \leq c(kb + 1)\epsilon |S^H| |S|, \quad |e_{ij}| \leq c(kb + 1)\epsilon \sqrt{b_{ii} b_{jj}}$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

The total number of floating-point operations for real flavors is approximately $n(kb+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that kb is much less than n .

After calling this routine, you can call [sbgst/hbgst](#) to solve the generalized eigenproblem $Az = \lambda Bz$, where A and B are banded and B is positive-definite.

Nonsymmetric Eigenvalue Problems: LAPACK Computational Routines

This topic describes LAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

A *nonsymmetric eigenvalue problem* is as follows: given a nonsymmetric (or non-Hermitian) matrix A , find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \text{ (right eigenvectors } z)$$

or the equation

$$z^H A = \lambda z^H \text{ (left eigenvectors } z).$$

Nonsymmetric eigenvalue problems have the following properties:

- The number of eigenvectors may be less than the matrix order (but is not less than the number of *distinct eigenvalues* of A).

- Eigenvalues may be complex even for a real matrix A .
- If a real nonsymmetric matrix has a complex eigenvalue $a+bi$ corresponding to an eigenvector z , then $a-bi$ is also an eigenvalue. The eigenvalue $a-bi$ corresponds to the eigenvector whose elements are complex conjugate to the elements of z .

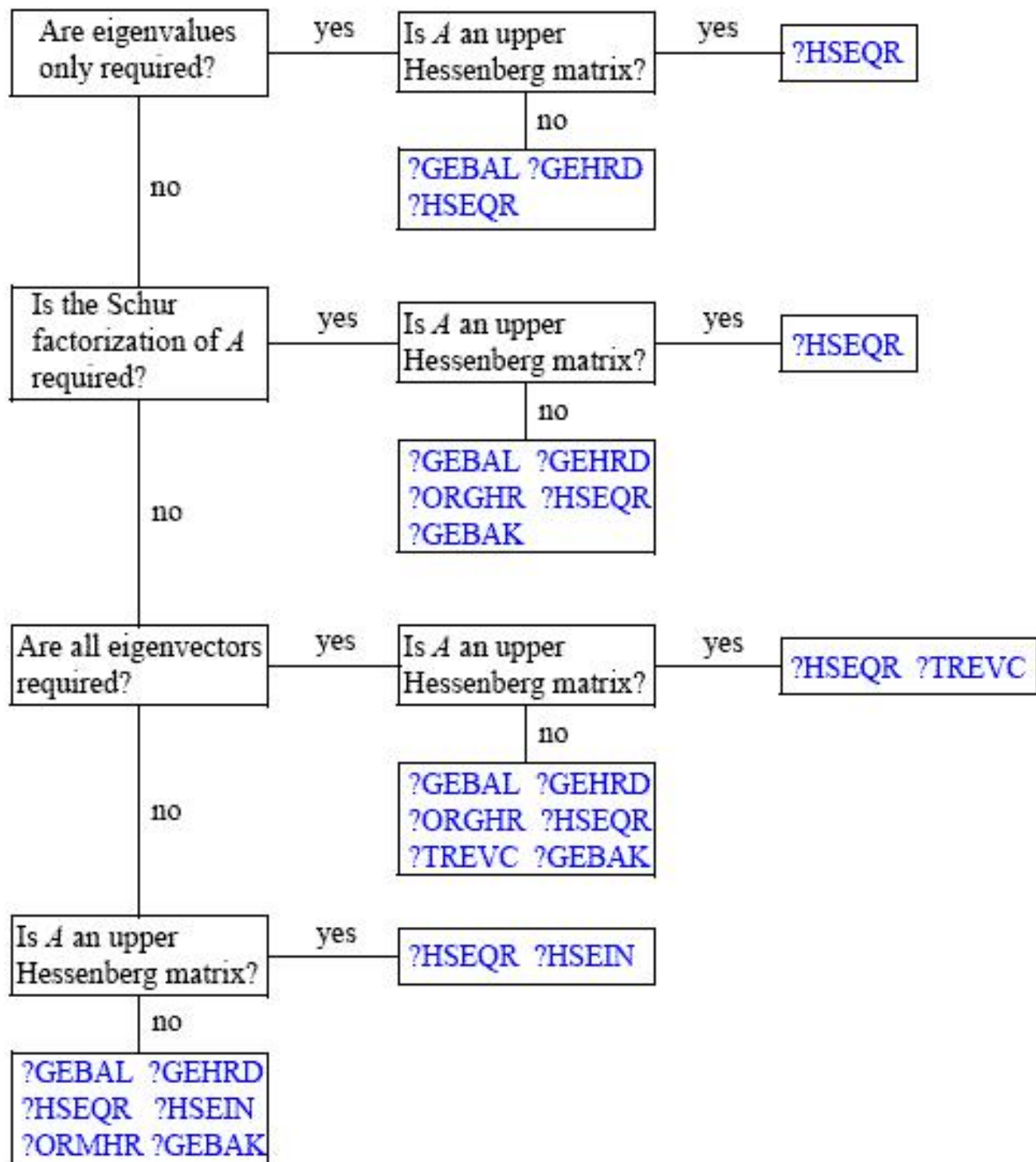
To solve a nonsymmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained. [Table "Computational Routines for Solving Nonsymmetric Eigenvalue Problems"](#) lists LAPACK routines to reduce the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$ as well as routines to solve eigenvalue problems with Hessenberg matrices, forming the Schur factorization of such matrices and computing the corresponding condition numbers.

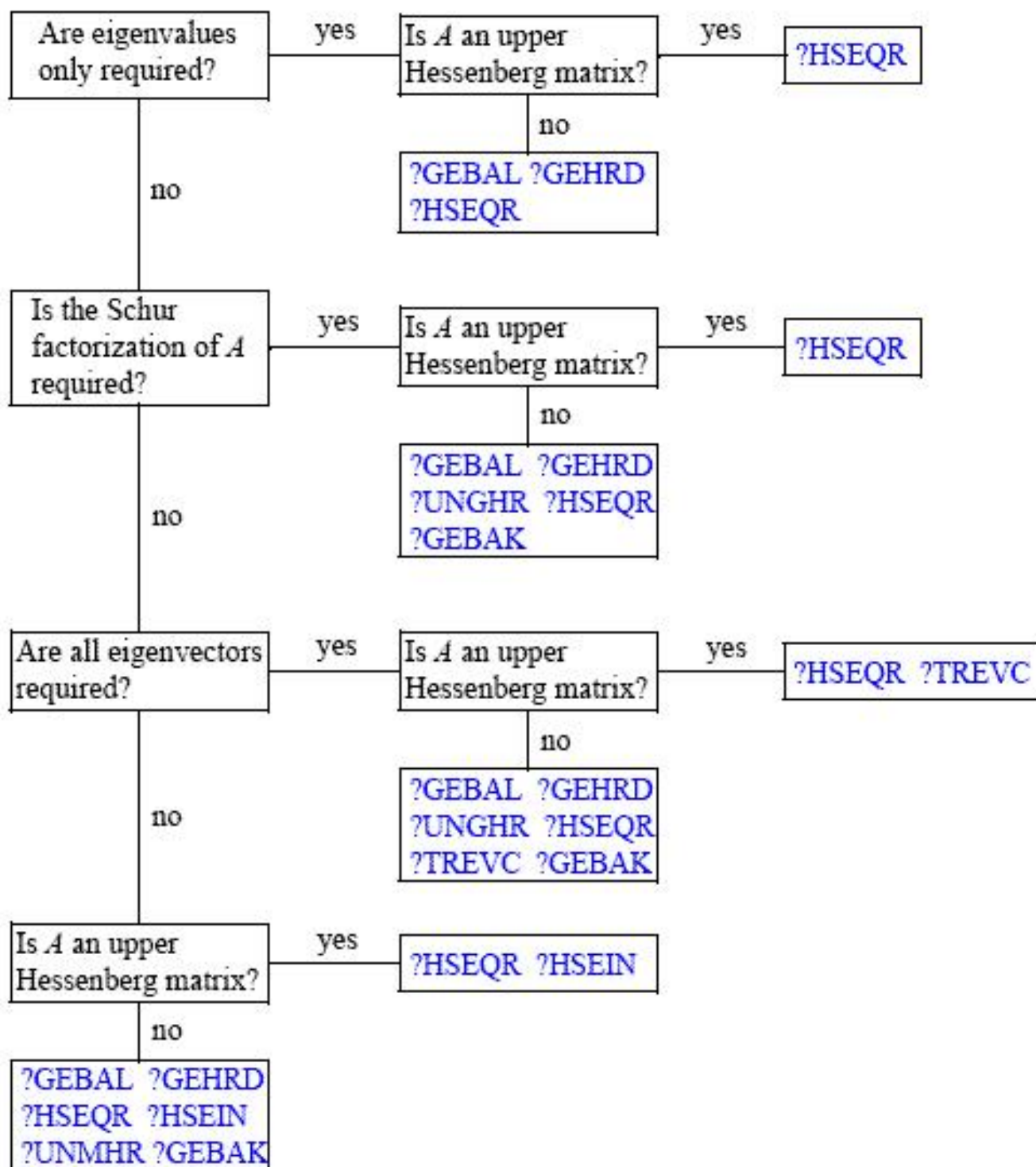
The decision tree in [Figure "Decision Tree: Real Nonsymmetric Eigenvalue Problems"](#) helps you choose the right routine or sequence of routines for an eigenvalue problem with a real nonsymmetric matrix. If you need to solve an eigenvalue problem with a complex non-Hermitian matrix, use the decision tree shown in [Figure "Decision Tree: Complex Non-Hermitian Eigenvalue Problems"](#).

Computational Routines for Solving Nonsymmetric Eigenvalue Problems

Operation performed	Routines for real matrices	Routines for complex matrices
Reduce to Hessenberg form $A = QHQ^H$?gehrd ,	?gehrd
Generate the matrix Q	?orghr	?unghr
Apply the matrix Q	?ormhr	?unmhr
Balance matrix	?gebal	?gebal
Transform eigenvectors of balanced matrix to those of the original matrix	?gebak	?gebak
Find eigenvalues and Schur factorization (QR algorithm)	?hseqr	?hseqr
Find eigenvectors from Hessenberg form (inverse iteration)	?hsein	?hsein
Find eigenvectors from Schur factorization	?trevc	?trevc
Estimate sensitivities of eigenvalues and eigenvectors	?trsna	?trsna
Reorder Schur factorization	?trexc	?trexc
Reorder Schur factorization, find the invariant subspace and estimate sensitivities	?trsen	?trsen
Solves Sylvester's equation.	?trsyl	?trsyl

Decision Tree: Real Nonsymmetric Eigenvalue Problems



Decision Tree: Complex Non-Hermitian Eigenvalue Problems*?gehrd*Reduces a general matrix to upper Hessenberg form.

Syntax

```
lapack_int LAPACKE_sgehrd (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, float* a, lapack_int lda, float* tau);
```

```
lapack_int LAPACKE_dgehrd (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, double* a, lapack_int lda, double* tau);
```

```
lapack_int LAPACKE_cgehrd (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);
```

```
lapack_int LAPACKE_zgehrd (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine reduces a general matrix A to upper Hessenberg form H by an orthogonal or unitary similarity transformation $A = Q^H H Q$. Here H has real subdiagonal elements.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>ilo</i> , <i>ihi</i>	If A is an output by ?gebal, then <i>ilo</i> and <i>ihi</i> must contain the values returned by that routine. Otherwise $ilo = 1$ and $ihi = n$. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)
<i>a</i>	Arrays: <i>a</i> (size $\max(1, lda * n)$) contains the matrix A .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	The elements on and above the subdiagonal contain the upper Hessenberg matrix H . The subdiagonal elements of H are real. The elements below the subdiagonal, with the array <i>tau</i> , represent the orthogonal matrix Q as a product of n elementary reflectors.
<i>tau</i>	Array, size at least $\max(1, n-1)$. Contains scalars that define elementary reflectors for the matrix Q .

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Application Notes

The computed Hessenberg matrix H is exactly similar to a nearby matrix $A + E$, where $\|E\|_2 < c(n)\epsilon\|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations for real flavors is $(2/3) * (ihi - ilo)^2 (2ihi + 2ilo + 3n)$; for complex flavors it is 4 times greater.

`?orghr`

Generates the real orthogonal matrix Q determined by `?gehrd`.

Syntax

```
lapack_int LAPACKE_sorghr (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int
ihi, float* a, lapack_int lda, const float* tau);
```

```
lapack_int LAPACKE_dorghr (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int
ihi, double* a, lapack_int lda, const double* tau);
```

Include Files

- `mkl.h`

Description

The routine explicitly generates the orthogonal matrix Q that has been determined by a preceding call to `sgehrd/dgehrd`. (The routine `?gehrd` reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^*H^*Q^T$, and represents the matrix Q as a product of ihi - ilo elementary reflectors. Here ilo and ihi are values determined by `sgebal/dgebal` when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

The matrix Q generated by `?orghr` has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns ilo to ihi .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>n</i>	The order of the matrix Q ($n \geq 0$).
<i>ilo, ihi</i>	These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to ?gehrd. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)
<i>a, tau</i>	Arrays: <i>a</i> (size $\max(1, lda * n)$) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd. <i>tau</i> contains further details of the elementary reflectors, as returned by ?gehrd. The dimension of <i>tau</i> must be at least $\max(1, n-1)$.
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	Overwritten by the n -by- n orthogonal matrix Q .
----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrix Q differs from the exact result by a matrix E such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3) (ihi-ilo)^3$.

The complex counterpart of this routine is [unghr](#).

?ormhr

Multiplies an arbitrary real matrix C by the real orthogonal matrix Q determined by ?gehrd.

Syntax

```
lapack_int LAPACKE_sormhr (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int ilo, lapack_int ihi, const float* a, lapack_int lda, const
float* tau, float* c, lapack_int ldc);
```

```
lapack_int LAPACKE_dormhr (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int ilo, lapack_int ihi, const double* a, lapack_int lda, const
double* tau, double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a matrix C by the orthogonal matrix Q that has been determined by a preceding call to `sgehrd/dgehrd`. (The routine `?gehrd` reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^T H Q$, and represents the matrix Q as a product of *ihilo* elementary reflectors. Here *ilo* and *ihi* are values determined by `sgebal/dgebal` when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

With `?ormhr`, you can form one of the matrix products $Q^T C$, $Q^T C$, $C^T Q$, or $C^T Q^T$, overwriting the result on C (which may be any real rectangular matrix).

A common application of `?ormhr` is to transform a matrix V of eigenvectors of H to the matrix QV of eigenvectors of A .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>side</code>	Must be 'L' or 'R'. If <code>side = 'L'</code> , then the routine forms $Q^T C$ or $Q^T C$. If <code>side = 'R'</code> , then the routine forms $C^T Q$ or $C^T Q^T$.
<code>trans</code>	Must be 'N' or 'T'. If <code>trans = 'N'</code> , then Q is applied to C . If <code>trans = 'T'</code> , then Q^T is applied to C .
<code>m</code>	The number of rows in C ($m \geq 0$).
<code>n</code>	The number of columns in C ($n \geq 0$).
<code>ilo, ihi</code>	These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to <code>?gehrd</code> . If $m > 0$ and <code>side = 'L'</code> , then $1 \leq ilo \leq ihi \leq m$. If $m = 0$ and <code>side = 'L'</code> , then $ilo = 1$ and $ihi = 0$. If $n > 0$ and <code>side = 'R'</code> , then $1 \leq ilo \leq ihi \leq n$. If $n = 0$ and <code>side = 'R'</code> , then $ilo = 1$ and $ihi = 0$.
<code>a, tau, c</code>	Arrays: a (size $\max(1, lda \cdot n)$ for <code>side='R'</code> and size $\max(1, lda \cdot m)$ for <code>side='L'</code>) contains details of the vectors which define the <i>elementary reflectors</i> , as returned by <code>?gehrd</code> . τ contains further details of the <i>elementary reflectors</i> , as returned by <code>?gehrd</code> . The dimension of τ must be at least $\max(1, m-1)$ if <code>side = 'L'</code> and at least $\max(1, n-1)$ if <code>side = 'R'</code> . c (size $\max(1, ldc \cdot n)$ for column major layout and $\max(1, ldc \cdot m)$ for row major layout) contains the m by n matrix C .

<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ if <i>side</i> = 'L' and at least $\max(1, n)$ if <i>side</i> = 'R'.
<i>ldc</i>	The leading dimension of <i>c</i> ; at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout .

Output Parameters

<i>c</i>	<i>C</i> is overwritten by product Q^*C , $Q^T C$, C^*Q , or $C^T Q$ as specified by <i>side</i> and <i>trans</i> .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrix *Q* differs from the exact result by a matrix *E* such that $\|E\|_2 = O(\epsilon) \|C\|_2$, where ϵ is the machine precision.

The approximate number of floating-point operations is

$2n(ihi-ilo)^2$ if *side* = 'L';

$2m(ihi-ilo)^2$ if *side* = 'R'.

The complex counterpart of this routine is [unmhr](#).

?unghr

Generates the complex unitary matrix Q determined by ?gehrd.

Syntax

```
lapack_int LAPACK_cunghr (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int
ihi, lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);
```

```
lapack_int LAPACK_zunghr (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int
ihi, lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine is intended to be used following a call to [cgehrd/zgehrd](#), which reduces a complex matrix *A* to upper Hessenberg form *H* by a unitary similarity transformation: $A = Q^*H^*Q^H$. *?gehrd* represents the matrix *Q* as a product of *ihi-ilo* elementary reflectors. Here *ilo* and *ihi* are values determined by [cgebal/zgebal](#) when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.

Use the routine [unghr](#) to generate *Q* explicitly as a square matrix. The matrix *Q* has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns ilo to ihi .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>n</code>	The order of the matrix Q ($n \geq 0$).
<code>ilo, ihi</code>	These must be the same parameters ilo and ihi , respectively, as supplied to <code>?gehrd</code> . (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$. If $n = 0$, then $ilo = 1$ and $ihi = 0$.)
<code>a, tau</code>	<p>Arrays:</p> <p>a (size $\max(1, lda*n)$) contains details of the vectors which define the <i>elementary reflectors</i>, as returned by <code>?gehrd</code>.</p> <p>tau contains further details of the <i>elementary reflectors</i>, as returned by <code>?gehrd</code>.</p> <p>The dimension of tau must be at least $\max(1, n-1)$.</p>
<code>lda</code>	The leading dimension of a ; at least $\max(1, n)$.

Output Parameters

<code>a</code>	Overwritten by the n -by- n unitary matrix Q .
----------------	--

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Application Notes

The computed matrix Q differs from the exact result by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of real floating-point operations is $(16/3)(ihi-ilo)^3$.

The real counterpart of this routine is [orghr](#).

?unmhr

Multiplies an arbitrary complex matrix C by the complex unitary matrix Q determined by ?gehrd.

Syntax

```
lapack_int LAPACKE_cunmhr (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int ilo, lapack_int ihi, const lapack_complex_float* a, lapack_int
lda, const lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);

lapack_int LAPACKE_zunmhr (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int ilo, lapack_int ihi, const lapack_complex_double* a,
lapack_int lda, const lapack_complex_double* tau, lapack_complex_double* c, lapack_int
ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a matrix C by the unitary matrix Q that has been determined by a preceding call to `cgehrd/zgehrd`. (The routine `?gehrd` reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^H H Q$, and represents the matrix Q as a product of *ihi-ilo elementary reflectors*. Here *ilo* and *ihi* are values determined by `cgebal/zgebal` when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.)

With `?unmhr`, you can form one of the matrix products $Q^H C$, $Q^H C$, $C^H Q$, or $C^H Q^H$, overwriting the result on C (which may be any complex rectangular matrix). A common application of this routine is to transform a matrix V of eigenvectors of H to the matrix QV of eigenvectors of A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	Must be 'L' or 'R'. If <i>side</i> = 'L', then the routine forms $Q^H C$ or $Q^H C$. If <i>side</i> = 'R', then the routine forms $C^H Q$ or $C^H Q^H$.
<i>trans</i>	Must be 'N' or 'C'. If <i>trans</i> = 'N', then Q is applied to C . If <i>trans</i> = 'T', then Q^H is applied to C .
<i>m</i>	The number of rows in C ($m \geq 0$).

<i>n</i>	The number of columns in <i>C</i> ($n \geq 0$).
<i>ilo, ihi</i>	These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to ?gehrd. If $m > 0$ and <i>side</i> = 'L', then $1 \leq ilo \leq ihi \leq m$. If $m = 0$ and <i>side</i> = 'L', then <i>ilo</i> = 1 and <i>ihi</i> = 0. If $n > 0$ and <i>side</i> = 'R', then $1 \leq ilo \leq ihi \leq n$. If $n = 0$ and <i>side</i> = 'R', then <i>ilo</i> = 1 and <i>ihi</i> = 0.
<i>a, tau, c</i>	Arrays: <i>a</i> (size max(1, <i>lda</i> * <i>n</i>) for <i>side</i> ='R' and size max(1, <i>lda</i> * <i>m</i>) for <i>side</i> ='L') contains details of the vectors which define the elementary reflectors, as returned by ?gehrd. <i>tau</i> contains further details of the elementary reflectors, as returned by ?gehrd. The dimension of <i>tau</i> must be at least max (1, <i>m</i> -1) if <i>side</i> = 'L' and at least max (1, <i>n</i> -1) if <i>side</i> = 'R'. <i>c</i> (size max(1, <i>ldc</i> * <i>n</i>) for column major layout and max(1, <i>ldc</i> * <i>m</i>) for row major layout) contains the <i>m</i> -by- <i>n</i> matrix <i>C</i> .
<i>lda</i>	The leading dimension of <i>a</i> ; at least max(1, <i>m</i>) if <i>side</i> = 'L' and at least max (1, <i>n</i>) if <i>side</i> = 'R'.
<i>ldc</i>	The leading dimension of <i>c</i> ; at least max(1, <i>m</i>) for column major layout and at least max(1, <i>n</i>) for row major layout.

Output Parameters

<i>c</i>	<i>C</i> is overwritten by Q^*C , or $Q^H C$, or $C Q^H$, or C^*Q as specified by <i>side</i> and <i>trans</i> .
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed matrix *Q* differs from the exact result by a matrix *E* such that $\|E\|_2 = O(\varepsilon) * \|C\|_2$, where ε is the machine precision.

The approximate number of floating-point operations is

$8n(ihi-ilo)^2$ if *side* = 'L';

$8m(ihi-ilo)^2$ if *side* = 'R'.

The real counterpart of this routine is [ormhr](#).

sgebal

Balances a general matrix to improve the accuracy of computed eigenvalues and eigenvectors.

Syntax

```

lapack_int LAPACKE_sgebal( int matrix_layout, char job, lapack_int n, float* a,
lapack_int lda, lapack_int* ilo, lapack_int* ihi, float* scale );

lapack_int LAPACKE_dgebal( int matrix_layout, char job, lapack_int n, double* a,
lapack_int lda, lapack_int* ilo, lapack_int* ihi, double* scale );

lapack_int LAPACKE_cgebal( int matrix_layout, char job, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_int* ilo, lapack_int* ihi, float*
scale );

lapack_int LAPACKE_zgebal( int matrix_layout, char job, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_int* ilo, lapack_int* ihi, double*
scale );

```

Include Files

- mkl.h

Description

The routine *balances* a matrix *A* by performing either or both of the following two similarity transformations:

(1) The routine first attempts to permute *A* to block upper triangular form:

$$PAP^T = A' = \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix}$$

where *P* is a permutation matrix, and *A'*₁₁ and *A'*₃₃ are upper triangular. The diagonal elements of *A'*₁₁ and *A'*₃₃ are eigenvalues of *A*. The rest of the eigenvalues of *A* are the eigenvalues of the central diagonal block *A'*₂₂, in rows and columns *ilo* to *ihi*. Subsequent operations to compute the eigenvalues of *A* (or its Schur factorization) need only be applied to these rows and columns; this can save a significant amount of work if *ilo* > 1 and *ihi* < *n*.

If no suitable permutation exists (as is often the case), the routine sets *ilo* = 1 and *ihi* = *n*, and *A'*₂₂ is the whole of *A*.

(2) The routine applies a diagonal similarity transformation to *A'*, to make the rows and columns of *A'*₂₂ as close in norm as possible:

$$A'' = DA'D^{-1} = \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & I \end{bmatrix} \times \begin{bmatrix} A'_{11} & A'_{12} & A'_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A'_{33} \end{bmatrix} \times \begin{bmatrix} I & 0 & 0 \\ 0 & D_{22}^{-1} & 0 \\ 0 & 0 & I \end{bmatrix}$$

This scaling can reduce the norm of the matrix (that is, $\|A''_{22}\| < \|A'_{22}\|$), and hence reduce the effect of rounding errors on the accuracy of computed eigenvalues and eigenvectors.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>job</i>	Must be 'N' or 'P' or 'S' or 'B'. If <i>job</i> = 'N', then <i>A</i> is neither permuted nor scaled (but <i>ilo</i> , <i>ihi</i> , and <i>scale</i> get their values). If <i>job</i> = 'P', then <i>A</i> is permuted but not scaled. If <i>job</i> = 'S', then <i>A</i> is scaled but not permuted. If <i>job</i> = 'B', then <i>A</i> is both scaled and permuted.
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	Array <i>a</i> (size $\max(1, lda*n)$) contains the matrix <i>A</i> .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	Overwritten by the balanced matrix (<i>a</i> is not referenced if <i>job</i> = 'N').
<i>ilo</i> , <i>ihi</i>	The values <i>ilo</i> and <i>ihi</i> such that on exit <i>a</i> (<i>i</i> , <i>j</i>) is zero if $i > j$ and $1 \leq j < ilo$ or $ihi < j \leq n$. If <i>job</i> = 'N' or 'S', then <i>ilo</i> = 1 and <i>ihi</i> = <i>n</i> .
<i>scale</i>	Array, size at least $\max(1, n)$. Contains details of the permutations and scaling factors. More precisely, if p_j is the index of the row and column interchanged with row and column <i>j</i> , and d_j is the scaling factor used to balance row and column <i>j</i> , then $scale[j - 1] = p_j$ for $j = 1, 2, \dots, ilo-1, ihi+1, \dots, n$; $scale[j - 1] = d_j$ for $j = ilo, ilo + 1, \dots, ihi$. The order in which the interchanges are made is <i>n</i> to <i>ihi</i> +1, then 1 to <i>ilo</i> -1.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The errors are negligible, compared with those in subsequent computations.

If the matrix A is balanced by this routine, then any eigenvectors computed subsequently are eigenvectors of the matrix A'' and hence you must call [gebak](#) to transform them back to eigenvectors of A .

If the Schur vectors of A are required, do not call this routine with $job = 'S'$ or $'B'$, because then the balancing transformation is not orthogonal (not unitary for complex flavors).

If you call this routine with $job = 'P'$, then any Schur vectors computed subsequently are Schur vectors of the matrix A'' , and you need to call [gebak](#) (with $side = 'R'$) to transform them back to Schur vectors of A .

The total number of floating-point operations is proportional to n^2 .

?gebak

Transforms eigenvectors of a balanced matrix to those of the original nonsymmetric matrix.

Syntax

```
lapack_int LAPACKE_sgebak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const float* scale, lapack_int m, float* v, lapack_int
ldv );

lapack_int LAPACKE_dgebak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const double* scale, lapack_int m, double* v,
lapack_int ldv );

lapack_int LAPACKE_cgebak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const float* scale, lapack_int m, lapack_complex_float*
v, lapack_int ldv );

lapack_int LAPACKE_zgebak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const double* scale, lapack_int m,
lapack_complex_double* v, lapack_int ldv );
```

Include Files

- mkl.h

Description

The routine is intended to be used after a matrix A has been balanced by a call to [?gebal](#), and eigenvectors of the balanced matrix A''_{22} have subsequently been computed. For a description of balancing, see [gebal](#). The balanced matrix A'' is obtained as $A'' = D * P * A * P^T * \text{inv}(D)$, where P is a permutation matrix and D is a diagonal scaling matrix. This routine transforms the eigenvectors as follows:

if x is a right eigenvector of A'' , then $P^T * \text{inv}(D) * x$ is a right eigenvector of A ; if y is a left eigenvector of A'' , then $P^T * D * y$ is a left eigenvector of A .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>job</i>	Must be 'N' or 'P' or 'S' or 'B'. The same parameter <i>job</i> as supplied to ?gebal .

<i>side</i>	Must be 'L' or 'R'. If <i>side</i> = 'L', then left eigenvectors are transformed. If <i>side</i> = 'R', then right eigenvectors are transformed.
<i>n</i>	The number of rows of the matrix of eigenvectors ($n \geq 0$).
<i>ilo, ihi</i>	The values <i>ilo</i> and <i>ihi</i> , as returned by ?gebal. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, then <i>ilo</i> = 1 and <i>ihi</i> = 0.)
<i>scale</i>	Array, size at least $\max(1, n)$. Contains details of the permutations and/or the scaling factors used to balance the original general matrix, as returned by ?gebal.
<i>m</i>	The number of columns of the matrix of eigenvectors ($m \geq 0$).
<i>v</i>	Arrays: <i>v</i> (size $\max(1, ldv*n)$ for column major layout and $\max(1, ldv*m)$ for row major layout) contains the matrix of left or right eigenvectors to be transformed.
<i>ldv</i>	The leading dimension of <i>v</i> ; at least $\max(1, n)$ for column major layout and at least $\max(1, m)$ for row major layout .

Output Parameters

<i>v</i>	Overwritten by the transformed eigenvectors.
----------	--

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The errors in this routine are negligible.

The approximate number of floating-point operations is approximately proportional to $m*n$.

?hseqr

Computes all eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.

Syntax

```
lapack_int LAPACKE_shseqr( int matrix_layout, char job, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, float* h, lapack_int ldh, float* wr, float* wi, float*
z, lapack_int ldz );
```

```
lapack_int LAPACKE_dhseqr( int matrix_layout, char job, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, double* h, lapack_int ldh, double* wr, double* wi,
double* z, lapack_int ldz );
```

```
lapack_int LAPACKE_chseqr( int matrix_layout, char job, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, lapack_complex_float* h, lapack_int ldh,
lapack_complex_float* w, lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhseqr( int matrix_layout, char job, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, lapack_complex_double* h, lapack_int ldh,
lapack_complex_double* w, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally the Schur factorization, of an upper Hessenberg matrix H : $H = Z^* T^* Z^H$, where T is an upper triangular (or, for real flavors, quasi-triangular) matrix (the Schur form of H), and Z is the unitary or orthogonal matrix whose columns are the Schur vectors z_i .

You can also use this routine to compute the Schur factorization of a general matrix A which has been reduced to upper Hessenberg form H :

$A = Q^* H^* Q^H$, where Q is unitary (orthogonal for real flavors);

$A = (QZ)^* T^* (QZ)^H$.

In this case, after reducing A to Hessenberg form by [gehrd](#), call [orghr](#) to form Q explicitly and then pass Q to `?hseqr` with `compz = 'V'`.

You can also call [gebal](#) to balance the original matrix before reducing it to Hessenberg form by `?hseqr`, so that the Hessenberg matrix H will have the structure:

$$\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ 0 & H_{22} & H_{23} \\ 0 & 0 & H_{33} \end{bmatrix}$$

where H_{11} and H_{33} are upper triangular.

If so, only the central diagonal block H_{22} (in rows and columns ilo to ihi) needs to be further reduced to Schur form (the blocks H_{12} and H_{23} are also affected). Therefore the values of ilo and ihi can be supplied to `?hseqr` directly. Also, after calling this routine you must call `gebak` to permute the Schur vectors of the balanced matrix to those of the original matrix.

If `?gebal` has not been called, however, then ilo must be set to 1 and ihi to n . Note that if the Schur factorization of A is required, `?gebal` must not be called with $job = 'S'$ or $'B'$, because the balancing transformation is not unitary (for real flavors, it is not orthogonal).

`?hseqr` uses a multishift form of the upper Hessenberg QR algorithm. The Schur vectors are normalized so that $\|z_i\|_2 = 1$, but are determined only to within a complex factor of absolute value 1 (for the real flavors, to within a factor ± 1).

Input Parameters

<i>job</i>	Must be 'E' or 'S'. If <i>job</i> = 'E', then eigenvalues only are required. If <i>job</i> = 'S', then the Schur form T is required.
<i>compz</i>	Must be 'N' or 'I' or 'V'. If <i>compz</i> = 'N', then no Schur vectors are computed (and the array z is not referenced). If <i>compz</i> = 'I', then the Schur vectors of H are computed (and the array z is initialized by the routine). If <i>compz</i> = 'V', then the Schur vectors of A are computed (and the array z must contain the matrix Q on entry).
<i>n</i>	The order of the matrix H ($n \geq 0$).
<i>ilo, ihi</i>	If A has been balanced by <code>?gebal</code> , then <i>ilo</i> and <i>ihi</i> must contain the values returned by <code>?gebal</code> . Otherwise, <i>ilo</i> must be set to 1 and <i>ihi</i> to n .
<i>h, z</i>	Arrays: h (size $\max(1, ldh*n)$) The n -by- n upper Hessenberg matrix H . z (size $\max(1, ldz*n)$) If <i>compz</i> = 'V', then z must contain the matrix Q from the reduction to Hessenberg form. If <i>compz</i> = 'I', then z need not be set. If <i>compz</i> = 'N', then z is not referenced.
<i>ldh</i>	The leading dimension of h ; at least $\max(1, n)$.
<i>ldz</i>	The leading dimension of z ; If <i>compz</i> = 'N', then $ldz \geq 1$. If <i>compz</i> = 'V' or 'I', then $ldz \geq \max(1, n)$.

Output Parameters

<i>w</i>	Array, size at least $\max(1, n)$. Contains the computed eigenvalues, unless <i>info</i> > 0. The eigenvalues are stored in the same order as on the diagonal of the Schur form T (if computed).
----------	---

wr, wi	<p>Arrays, size at least $\max(1, n)$ each.</p> <p>Contain the real and imaginary parts, respectively, of the computed eigenvalues, unless $info > 0$. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first. The eigenvalues are stored in the same order as on the diagonal of the Schur form T (if computed).</p>
h	<p>If $info = 0$ and $job = 'S'$, h contains the upper quasi-triangular matrix T from the Schur decomposition (the Schur form).</p> <p>If $info = 0$ and $job = 'E'$, the contents of h are unspecified on exit. (The output value of h when $info > 0$ is given under the description of $info$ below.)</p>
z	<p>If $compz = 'V'$ and $info = 0$, then z contains Q^*Z.</p> <p>If $compz = 'I'$ and $info = 0$, then z contains the unitary or orthogonal matrix Z of the Schur vectors of H.</p> <p>If $compz = 'N'$, then z is not referenced.</p>

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, `?hseqr` failed to compute all of the eigenvalues. Elements 1,2, ..., $ilo-1$ and $i+1, i+2, \dots, n$ of the eigenvalue arrays (wr and wi for real flavors and w for complex flavors) contain the real and imaginary parts of those eigenvalues that have been successfully found.

If $info > 0$, and $job = 'E'$, then on exit, the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns ilo through $info$ of the final output value of H .

If $info > 0$, and $job = 'S'$, then on exit $(\text{initial value of } H) * U = U * (\text{final value of } H)$, where U is a unitary matrix. The final value of H is upper Hessenberg and triangular in rows and columns $info+1$ through ihl .

If $info > 0$, and $compz = 'V'$, then on exit $(\text{final value of } Z) = (\text{initial value of } Z) * U$, where U is the unitary matrix (regardless of the value of job).

If $info > 0$, and $compz = 'I'$, then on exit $(\text{final value of } Z) = U$, where U is the unitary matrix (regardless of the value of job).

If $info > 0$, and $compz = 'N'$, then Z is not accessed.

Application Notes

The computed Schur factorization is the exact factorization of a nearby matrix $H + E$, where $\|E\|_2 < O(\epsilon) \|H\|_2 / s_i$, and ϵ is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then $|\lambda_i - \mu_i| \leq c(n) * \epsilon * \|H\|_2 / s_i$, where $c(n)$ is a modestly increasing function of n , and s_i is the reciprocal condition number of λ_i . The condition numbers s_i may be computed by calling [trсна](#).

The total number of floating-point operations depends on how rapidly the algorithm converges; typical numbers are as follows.

If only eigenvalues are computed:

- $7n^3$ for real flavors
- $25n^3$ for complex flavors.

If the Schur form is computed:	$10n^3$ for real flavors
	$35n^3$ for complex flavors.
If the full Schur factorization is computed:	$20n^3$ for real flavors
	$70n^3$ for complex flavors.

?hsein

Computes selected eigenvectors of an upper Hessenberg matrix that correspond to specified eigenvalues.

Syntax

```
lapack_int LAPACKE_shsein( int matrix_layout, char side, char eigsrc, char initv,
lapack_logical* select, lapack_int n, const float* h, lapack_int ldh, float* wr, const
float* wi, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr, lapack_int mm,
lapack_int* m, lapack_int* ifaill, lapack_int* ifailr );
```

```
lapack_int LAPACKE_dhsein( int matrix_layout, char side, char eigsrc, char initv,
lapack_logical* select, lapack_int n, const double* h, lapack_int ldh, double* wr,
const double* wi, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr, lapack_int
mm, lapack_int* m, lapack_int* ifaill, lapack_int* ifailr );
```

```
lapack_int LAPACKE_chsein( int matrix_layout, char side, char eigsrc, char initv, const
lapack_logical* select, lapack_int n, const lapack_complex_float* h, lapack_int ldh,
lapack_complex_float* w, lapack_complex_float* vl, lapack_int ldvl,
lapack_complex_float* vr, lapack_int ldvr, lapack_int mm, lapack_int* m, lapack_int*
ifaill, lapack_int* ifailr );
```

```
lapack_int LAPACKE_zhsein( int matrix_layout, char side, char eigsrc, char initv, const
lapack_logical* select, lapack_int n, const lapack_complex_double* h, lapack_int ldh,
lapack_complex_double* w, lapack_complex_double* vl, lapack_int ldvl,
lapack_complex_double* vr, lapack_int ldvr, lapack_int mm, lapack_int* m, lapack_int*
ifaill, lapack_int* ifailr );
```

Include Files

- mkl.h

Description

The routine computes left and/or right eigenvectors of an upper Hessenberg matrix H , corresponding to selected eigenvalues.

The right eigenvector x and the left eigenvector y , corresponding to an eigenvalue λ , are defined by: $H^*x = \lambda^*x$ and $y^H H = \lambda^* y^H$ (or $H^H y = \lambda^* y$). Here λ^* denotes the conjugate of λ .

The eigenvectors are computed by inverse iteration. They are scaled so that, for a real eigenvector x , $\max |x_i| = 1$, and for a complex eigenvector, $\max (|\operatorname{Re} x_i| + |\operatorname{Im} x_i|) = 1$.

If H has been formed by reduction of a general matrix A to upper Hessenberg form, then eigenvectors of H may be transformed to eigenvectors of A by [ormhr](#) or [unmhr](#).

Input Parameters

<code>side</code>	<p>Must be 'R' or 'L' or 'B'.</p> <p>If <code>side</code> = 'R', then only right eigenvectors are computed.</p> <p>If <code>side</code> = 'L', then only left eigenvectors are computed.</p> <p>If <code>side</code> = 'B', then all eigenvectors are computed.</p>
<code>eigsrc</code>	<p>Must be 'Q' or 'N'.</p> <p>If <code>eigsrc</code> = 'Q', then the eigenvalues of H were found using hseqr; thus if H has any zero sub-diagonal elements (and so is block triangular), then the j-th eigenvalue can be assumed to be an eigenvalue of the block containing the j-th row/column. This property allows the routine to perform inverse iteration on just one diagonal block. If <code>eigsrc</code> = 'N', then no such assumption is made and the routine performs inverse iteration using the whole matrix.</p>
<code>initv</code>	<p>Must be 'N' or 'U'.</p> <p>If <code>initv</code> = 'N', then no initial estimates for the selected eigenvectors are supplied.</p> <p>If <code>initv</code> = 'U', then initial estimates for the selected eigenvectors are supplied in <code>vl</code> and/or <code>vr</code>.</p>
<code>select</code>	<p>Array, size at least $\max(1, n)$. Specifies which eigenvectors are to be computed.</p> <p><i>For real flavors:</i></p> <p>To obtain the real eigenvector corresponding to the real eigenvalue <code>wr[j]</code>, set <code>select[j]</code> to 1</p> <p>To select the complex eigenvector corresponding to the complex eigenvalue (<code>wr[j - 1]</code>, <code>wi[j - 1]</code>) with complex conjugate (<code>wr[j]</code>, <code>wi[j]</code>), set <code>select[j - 1]</code> and/or <code>select[j]</code> to 1; the eigenvector corresponding to the first eigenvalue in the pair is computed.</p> <p><i>For complex flavors:</i></p> <p>To select the eigenvector corresponding to the eigenvalue <code>w[j]</code>, set <code>select[j]</code> to 1</p>
<code>n</code>	The order of the matrix H ($n \geq 0$).
<code>h</code> , <code>vl</code> , <code>vr</code>	<p>Arrays:</p> <p><code>h</code> (size $\max(1, ldh*n)$) The n-by-n upper Hessenberg matrix H. If an NaN value is detected in <code>h</code>, the routine returns with <code>info</code> = -6.</p> <p><code>vl</code>(size $\max(1, ldvl*mm)$ for column major layout and $\max(1, ldvl*n)$ for row major layout)</p> <p>If <code>initv</code> = 'V' and <code>side</code> = 'L' or 'B', then <code>vl</code> must contain starting vectors for inverse iteration for the left eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.</p> <p>If <code>initv</code> = 'N', then <code>vl</code> need not be set.</p>

The array *vl* is not referenced if *side* = 'R'.

vr(size max(1, *ldvr***mm*) for column major layout and max(1, *ldvr***n*) for row major layout)

If *initv* = 'V' and *side* = 'R' or 'B', then *vr* must contain starting vectors for inverse iteration for the right eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.

If *initv* = 'N', then *vr* need not be set.

The array *vr* is not referenced if *side* = 'L'.

ldh

The leading dimension of *h*; at least max(1, *n*).

w

Array, size at least max(1, *n*).

Contains the eigenvalues of the matrix *H*.

If *eigsrc* = 'Q', the array must be exactly as returned by ?hseqr.

wr, wi

Arrays, size at least max(1, *n*) each.

Contain the real and imaginary parts, respectively, of the eigenvalues of the matrix *H*. Complex conjugate pairs of values must be stored in consecutive elements of the arrays. If *eigsrc* = 'Q', the arrays must be exactly as returned by ?hseqr.

ldvl

The leading dimension of *vl*.

If *side* = 'L' or 'B', *ldvl* ≥ max(1, *n*) for column major layout and *ldvl* ≥ max(1, *mm*) for row major layout .

If *side* = 'R', *ldvl* ≥ 1.

ldvr

The leading dimension of *vr*.

If *side* = 'R' or 'B', *ldvr* ≥ max(1, *n*) for column major layout and *ldvr* ≥ max(1, *mm*) for row major layout .

If *side* = 'L', *ldvr* ≥ 1.

mm

The number of columns in *vl* and/or *vr*.

Must be at least *m*, the actual number of columns required (see *Output Parameters* below).

For real flavors, *m* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector (see *select*).

For complex flavors, *m* is the number of selected eigenvectors (see *select*).

Constraint:

$$0 \leq mm \leq n.$$

Output Parameters

select

Overwritten for real flavors only.

If a complex eigenvector was selected as specified above, then *select*[*j* - 1] is set to 1 and *select*[*j*] to 0

<i>w</i>	The real parts of some elements of <i>w</i> may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.
<i>wr</i>	Some elements of <i>wr</i> may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.
<i>vl</i> , <i>vr</i>	<p>If <i>side</i> = 'L' or 'B', <i>vl</i> contains the computed left eigenvectors (as specified by <i>select</i>).</p> <p>If <i>side</i> = 'R' or 'B', <i>vr</i> contains the computed right eigenvectors (as specified by <i>select</i>).</p> <p>The eigenvectors treated column-wise form a rectangular <i>n</i>-by-<i>mm</i> matrix.</p> <p><i>For real flavors:</i> a real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns: the first column holds the real part of the eigenvector and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by <i>matrix_layout</i> (using either column major or row major layout).</p>
<i>m</i>	<p><i>For real flavors:</i> the number of columns of <i>vl</i> and/or <i>vr</i> required to store the selected eigenvectors.</p> <p><i>For complex flavors:</i> the number of selected eigenvectors.</p>
<i>ifaill</i> , <i>ifailr</i>	<p>Arrays, size at least $\max(1, mm)$ each.</p> <p><i>ifaill</i>[<i>i</i> - 1] = 0 if the <i>i</i>th column of <i>vl</i> converged;</p> <p><i>ifaill</i>[<i>i</i> - 1] = <i>j</i> > 0 if the eigenvector stored in the <i>i</i>-th column of <i>vl</i> (corresponding to the <i>j</i>th eigenvalue) failed to converge.</p> <p><i>ifailr</i>[<i>i</i> - 1] = 0 if the <i>i</i>th column of <i>vr</i> converged;</p> <p><i>ifailr</i>[<i>i</i> - 1] = <i>j</i> > 0 if the eigenvector stored in the <i>i</i>-th column of <i>vr</i> (corresponding to the <i>j</i>th eigenvalue) failed to converge.</p> <p><i>For real flavors:</i> if the <i>i</i>th and (<i>i</i>+1)th columns of <i>vl</i> contain a selected complex eigenvector, then <i>ifaill</i>[<i>i</i> - 1] and <i>ifaill</i>[<i>i</i>] are set to the same value. A similar rule holds for <i>vr</i> and <i>ifailr</i>.</p> <p>The array <i>ifaill</i> is not referenced if <i>side</i> = 'R'. The array <i>ifailr</i> is not referenced if <i>side</i> = 'L'.</p>

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, then *i* eigenvectors (as indicated by the parameters *ifaill* and/or *ifailr* above) failed to converge. The corresponding columns of *vl* and/or *vr* contain no useful information.

Application Notes

Each computed right eigenvector x_i is the exact eigenvector of a nearby matrix $A + E_i$, such that $\|E_i\| < O(\epsilon) \|A\|$. Hence the residual is small:

$$\|Ax_i - \lambda_i x_i\| = O(\epsilon) \|A\|.$$

However, eigenvectors corresponding to close or coincident eigenvalues may not accurately span the relevant subspaces.

Similar remarks apply to computed left eigenvectors.

?trevc

Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by ?hseqr.

Syntax

```
lapack_int LAPACKE_strevc( int matrix_layout, char side, char howmny, lapack_logical*
select, lapack_int n, const float* t, lapack_int ldt, float* vl, lapack_int ldvl, float*
vr, lapack_int ldvr, lapack_int mm, lapack_int* m );
```

```
lapack_int LAPACKE_dtrevc( int matrix_layout, char side, char howmny, lapack_logical*
select, lapack_int n, const double* t, lapack_int ldt, double* vl, lapack_int ldvl,
double* vr, lapack_int ldvr, lapack_int mm, lapack_int* m );
```

```
lapack_int LAPACKE_ctrevc( int matrix_layout, char side, char howmny, const
lapack_logical* select, lapack_int n, lapack_complex_float* t, lapack_int ldt,
lapack_complex_float* vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr,
lapack_int mm, lapack_int* m );
```

```
lapack_int LAPACKE_ztrevc( int matrix_layout, char side, char howmny, const
lapack_logical* select, lapack_int n, lapack_complex_double* t, lapack_int ldt,
lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int ldvr,
lapack_int mm, lapack_int* m );
```

Include Files

- mkl.h

Description

The routine computes some or all of the right and/or left eigenvectors of an upper triangular matrix T (or, for real flavors, an upper quasi-triangular matrix T). Matrices of this type are produced by the Schur factorization of a general matrix: $A = Q^* T^* Q^H$, as computed by [hseqr](#).

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w , are defined by:

$$T^* x = w^* x, \quad y^H T = w^* y^H, \quad \text{where } y^H \text{ denotes the conjugate transpose of } y.$$

The eigenvalues are not input to this routine, but are read directly from the diagonal blocks of T .

This routine returns the matrices X and/or Y of right and left eigenvectors of T , or the products $Q^* X$ and/or $Q^* Y$, where Q is an input matrix.

If Q is the orthogonal/unitary factor that reduces a matrix A to Schur form T , then $Q^* X$ and $Q^* Y$ are the matrices of right and left eigenvectors of A .

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

side Must be 'R' or 'L' or 'B'.

If *side* = 'R', then only right eigenvectors are computed.

	<p>If <i>side</i> = 'L', then only left eigenvectors are computed.</p> <p>If <i>side</i> = 'B', then all eigenvectors are computed.</p>
<i>howmny</i>	<p>Must be 'A' or 'B' or 'S'.</p> <p>If <i>howmny</i> = 'A', then all eigenvectors (as specified by <i>side</i>) are computed.</p> <p>If <i>howmny</i> = 'B', then all eigenvectors (as specified by <i>side</i>) are computed and backtransformed by the matrices supplied in <i>vl</i> and <i>vr</i>.</p> <p>If <i>howmny</i> = 'S', then selected eigenvectors (as specified by <i>side</i> and <i>select</i>) are computed.</p>
<i>select</i>	<p>Array, size at least max (1, <i>n</i>).</p> <p>If <i>howmny</i> = 'S', <i>select</i> specifies which eigenvectors are to be computed.</p> <p>If <i>howmny</i> = 'A' or 'B', <i>select</i> is not referenced.</p> <p><i>For real flavors:</i></p> <p>If <i>omega</i>[<i>j</i>] is a real eigenvalue, the corresponding real eigenvector is computed if <i>select</i>[<i>j</i>] is 1.</p> <p>If <i>omega</i>[<i>j</i> - 1] and <i>omega</i>[<i>j</i>] are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either <i>select</i>[<i>j</i> - 1] or <i>select</i>[<i>j</i>] is 1, and on exit <i>select</i>[<i>j</i> - 1] is set to 1 and <i>select</i>[<i>j</i>] is set to 0.</p> <p><i>For complex flavors:</i></p> <p>The eigenvector corresponding to the <i>j</i>-th eigenvalue is computed if <i>select</i>[<i>j</i> - 1] is 1.</p>
<i>n</i>	<p>The order of the matrix <i>T</i> (<i>n</i> ≥ 0).</p>
<i>t, vl, vr</i>	<p>Arrays:</p> <p><i>t</i> (size max(1, <i>ldt</i>*<i>n</i>)) contains the <i>n</i>-by-<i>n</i> matrix <i>T</i> in Schur canonical form. For complex flavors <i>ctrevc</i> and <i>ztrevc</i>, contains the upper triangular matrix <i>T</i>.</p> <p><i>vl</i>(size max(1, <i>ldvl</i>*<i>mm</i>) for column major layout and max(1, <i>ldvl</i>*<i>n</i>) for row major layout)</p> <p>If <i>howmny</i> = 'B' and <i>side</i> = 'L' or 'B', then <i>vl</i> must contain an <i>n</i>-by-<i>n</i> matrix <i>Q</i> (usually the matrix of Schur vectors returned by <i>?hseqr</i>).</p> <p>If <i>howmny</i> = 'A' or 'S', then <i>vl</i> need not be set.</p> <p>The array <i>vl</i> is not referenced if <i>side</i> = 'R'.</p> <p><i>vr</i>(size max(1, <i>ldvr</i>*<i>mm</i>) for column major layout and max(1, <i>ldvr</i>*<i>n</i>) for row major layout)</p> <p>If <i>howmny</i> = 'B' and <i>side</i> = 'R' or 'B', then <i>vr</i> must contain an <i>n</i>-by-<i>n</i> matrix <i>Q</i> (usually the matrix of Schur vectors returned by <i>?hseqr</i>). .</p> <p>If <i>howmny</i> = 'A' or 'S', then <i>vr</i> need not be set.</p> <p>The array <i>vr</i> is not referenced if <i>side</i> = 'L'.</p>

<i>ldt</i>	The leading dimension of <i>t</i> ; at least $\max(1, n)$.
<i>ldvl</i>	<p>The leading dimension of <i>vl</i>.</p> <p>If <i>side</i> = 'L' or 'B', $ldvl \geq n$.</p> <p>If <i>side</i> = 'R', $ldvl \geq 1$.</p>
<i>ldvr</i>	<p>The leading dimension of <i>vr</i>.</p> <p>If <i>side</i> = 'R' or 'B', $ldvr \geq n$.</p> <p>If <i>side</i> = 'L', $ldvr \geq 1$.</p>
<i>mm</i>	<p>The number of columns in the arrays <i>vl</i> and/or <i>vr</i>. Must be at least <i>m</i> (the precise number of columns required).</p> <p>If <i>howmny</i> = 'A' or 'B', $mm = n$.</p> <p>If <i>howmny</i> = 'S': for real flavors, <i>mm</i> is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector; for complex flavors, <i>mm</i> is the number of selected eigenvectors (see <i>select</i>).</p> <p>Constraint: $0 \leq mm \leq n$.</p>

Output Parameters

<i>select</i>	If a complex eigenvector of a real matrix was selected as specified above, then <i>select</i> [<i>j</i>] is set to 1 and <i>select</i> [<i>j</i> + 1] to 0
<i>t</i>	<i>ctrevc</i> / <i>ztrevc</i> modify the <i>t</i> array, which is restored on exit.
<i>vl, vr</i>	<p>If <i>side</i> = 'L' or 'B', <i>vl</i> contains the computed left eigenvectors (as specified by <i>howmny</i> and <i>select</i>).</p> <p>If <i>side</i> = 'R' or 'B', <i>vr</i> contains the computed right eigenvectors (as specified by <i>howmny</i> and <i>select</i>).</p> <p>The eigenvectors treated column-wise form a rectangular <i>n</i>-by-<i>mm</i> matrix.</p> <p><i>For real flavors</i>: a real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns: the first column holds the real part of the eigenvector and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by <i>matrix_layout</i> (using either column major or row major layout).</p>
<i>m</i>	<p><i>For complex flavors</i>: the number of selected eigenvectors.</p> <p>If <i>howmny</i> = 'A' or 'B', <i>m</i> is set to <i>n</i>.</p> <p><i>For real flavors</i>: the number of columns of <i>vl</i> and/or <i>vr</i> actually used to store the selected eigenvectors.</p> <p>If <i>howmny</i> = 'A' or 'B', <i>m</i> is set to <i>n</i>.</p>

Return Values

This function returns a value *info*.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

Application Notes

If x_i is an exact right eigenvector and y_i is the corresponding computed eigenvector, then the angle $\theta(y_i, x_i)$ between them is bounded as follows: $\theta(y_i, x_i) \leq (c(n)\varepsilon \|T\|_2) / \text{sep}_i$ where sep_i is the reciprocal condition number of x_i . The condition number sep_i may be computed by calling `?trsna`.

`?trevc3`

Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by ?hseqr using Level 3 BLAS

Syntax

```
call strevc3(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, lwork, info)
```

```
call dtrevc3(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, lwork, info)
```

```
call ctrevc3(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, lwork, rwork, lrwork, info)
```

```
call ztrevc3(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, lwork, rwork, lrwork, info)
```

Include Files

- `mkl.fi`

Description

This routine computes some or all of the right and left eigenvectors of an upper triangular matrix T (or, for real flavors, an upper quasi-triangular matrix T) using Level 3 BLAS. Matrices of this type are produced by the Schur factorization of a general matrix: $A = Q^*T^*QH$, as computed by `hseqr`.

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w are defined by the following:

$$T^*x = w^*x, \quad y^H T = w^*y^H$$

where y^H denotes the conjugate transpose of y .

The eigenvalues are not passed to this routine but are read directly from the diagonal blocks of T .

This routine returns one or both of the matrices X and Y of the right and left eigenvectors of T , or one or both of the products Q^*X and Q^*Y , where Q is an input matrix.

If Q is the orthogonal/unitary factor that reduces a matrix A to Schur form T , then Q^*X and Q^*Y are the matrices of the right and left eigenvectors of A .

Input Parameters

`side`

CHARACTER*1

Must be 'R', 'L', or 'B'.

- If `side = 'R'`, only right eigenvectors are computed.
- If `side = 'L'`, only left eigenvectors are computed.

	<ul style="list-style-type: none"> • If <i>side</i> = 'B', all eigenvectors are computed.
<i>howmny</i>	CHARACTER*1 Must be 'A', 'B', or 'S'. <ul style="list-style-type: none"> • If <i>howmny</i> = 'A', all eigenvectors (as specified by <i>side</i>) are computed. • If <i>howmny</i> = 'B', all eigenvectors (as specified by <i>side</i>) are computed and back-transformed by the matrices supplied in <i>vl</i> and <i>vr</i>. • If <i>howmny</i> = 'S', selected eigenvectors (as specified by <i>side</i> and <i>select</i>) are computed.
<i>select</i>	Array with a size of at least $\max(1, n)$ If <i>howmny</i> = 'S', <i>select</i> specifies which eigenvectors are to be computed. If <i>howmny</i> = 'A' or <i>howmny</i> = 'B', <i>select</i> is not referenced. For real flavors: <ul style="list-style-type: none"> • If $\omega(j)$ is a real eigenvalue and <i>select</i>(<i>j</i>) is .TRUE., the corresponding real eigenvector is computed. • If $\omega(j)$ and $\omega(j + 1)$ are the real and imaginary parts of a complex eigenvalue and either <i>select</i>(<i>j</i>) or <i>select</i>(<i>j + 1</i>) is .TRUE., the corresponding complex eigenvector is computed, and on exit <i>select</i>(<i>j</i>) is set to .TRUE. and <i>select</i>(<i>j + 1</i>) is set to .FALSE.. For complex flavors: <ul style="list-style-type: none"> • If <i>select</i>(<i>j</i>) is .TRUE., the eigenvector corresponding to the <i>j</i>th eigenvalue is computed.
<i>n</i>	INTEGER The order of the matrix <i>T</i> ($n \geq 0$).
<i>t, vl, vr, work</i>	<ul style="list-style-type: none"> • REAL for <i>strevc3</i> • DOUBLE PRECISION for <i>dtrevc3</i> • COMPLEX for <i>ctrevc3</i> • DOUBLE COMPLEX for <i>ztrevc3</i> Arrays: <ul style="list-style-type: none"> • <i>t</i>(<i>ldt</i>,*) contains the <i>n</i>-by-<i>n</i> matrix <i>T</i> in Schur canonical form. For complex flavors <i>ctrevc3</i> and <i>ztrevc3</i>, the array contains the upper triangular matrix <i>T</i>. The second dimension of <i>t</i> must be at least $\max(1, n)$. • <i>vl</i>(<i>ldvl</i>,*) If <i>howmny</i> = 'B' and <i>side</i> = 'L' or 'B', then <i>vl</i> must contain an <i>n</i>-by-<i>n</i> matrix <i>Q</i> (usually the matrix of Schur vectors returned by ?hseqr). If <i>howmny</i> = 'A' or 'S', <i>vl</i> need not be set.

The second dimension of *vl* must be at least $\max(1, \text{mm})$ if *side* = 'L' or 'B', and at least 1 if *side* = 'R'.

The array *vl* is not referenced if *side* = 'R'.

- *vr(ldvr,*)*

If *howmny* = 'B' and *side* = 'R' or 'B', *vr* must contain an *n*-by-*n* matrix *Q* (usually the matrix of Schur vectors returned by ?hseqr).

If *howmny* = 'A' or 'S', *vr* need not be set.

The second dimension of *vr* must be at least $\max(1, \text{mm})$ if *side* = 'R' or 'B', and at least 1 if *side* = 'L'.

The array *vr* is not referenced if *side* = 'L'.

- *work(*)* is a workspace array, and its dimension is $\max(1, \text{lwork})$.

lwork

INTEGER

The size of the work array. Must be at least $\max(1, 3*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

If *lwork* = -1, a workspace query is assumed; the routine calculates only the optimal size of the work array and returns this value as the first entry of the work array, and no error message related to *lwork* is issued by xerbla. For details, see "Application Notes" below.

ldt

INTEGER

The leading dimension of *t*. It is at least $\max(1, n)$.

ldvl

INTEGER

The leading dimension of *vl*.

- If *side* = 'L' or 'B', $\text{ldvl} \geq n$.
- If *side* = 'R', $\text{ldvl} \geq 1$.

ldvr

INTEGER

The leading dimension of *vr*.

- If *side* = 'R' or 'B', $\text{ldvr} \geq n$.
- If *side* = 'L', $\text{ldvr} \geq 1$.

mm

INTEGER

The number of columns in one or both of the arrays *vl* and *vr*. Must be at least *m* (the precise number of columns required).

- If *howmny* = 'A' or 'B', $\text{mm} = n$.
- If *howmny* = 'S': for real flavors, *mm* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector; for complex flavors, *mm* is the number of selected eigenvectors (see *select*).

Constraint: $0 \leq \text{mm} \leq n$.

rwork

- REAL for ctrevc3

- DOUBLE PRECISION for `ztrevc3`

The workspace array is used in complex flavors only. Its dimension is $\max(1, \text{lrwork})$.

lrwork

INTEGER

The size of the *rwork* array. It must be at least $\max(1, n)$.

If *lrwork* = -1, a workspace query is assumed; the routine calculates only the optimal size of the work array and returns this value as the first entry of the *rwork* array, and no error message related to *lrwork* is issued by xerbla. For details, see "Application Notes" below.

Output Parameters

select

If a complex eigenvector of a real matrix was selected as specified above, then `select(j)` is set to `.TRUE.` and `select(j + 1)` is set to `.FALSE.`.

t

COMPLEX for `ctrevc3`

DOUBLE COMPLEX for `ztrevc3`

`ctrevc3` or `ztrevc3` modifies the `t(ldt,*)` array, which is restored on exit.

vl, vr

If *side* = 'L' or 'B', *vl* contains the computed left eigenvectors (as specified by *howmny* and *select*).

If *side* = 'R' or 'B', *vr* contains the computed right eigenvectors (as specified by *howmny* and *select*).

Treated column-wise, the eigenvectors form a rectangular *n*-by-*mm* matrix.

For real flavors A real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns. The first column holds the real part of the eigenvector, and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by `matrix_layout` (using either column major or row major layout).

m

INTEGER

For complex flavors The number of selected eigenvectors. If *howmny* = 'A' or 'B', *m* is set to *n*.

For real flavors The number of columns of one or both of *vl* and *vr* actually used to store the selected eigenvectors. If *howmny* = 'A' or 'B', *m* is set to *n*.

<code>work(1)</code>	On exit, if <code>info = 0</code> , <code>work(1)</code> returns the required optimal size of <code>lwork</code> .
<code>rwork(1)</code>	On exit, if <code>info = 0</code> , then <code>rwork(1)</code> returns the required optimal size of <code>lrwork</code> .
<code>info</code>	INTEGER If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the i^{th} parameter contained an illegal value.

Application Notes

If x_i is an exact right eigenvector and y_i is the corresponding computed eigenvector, the angle $\theta(y_i, x_i)$ between them is bounded as follows:

$$\theta(y_i, x_i) \leq (c(n) \varepsilon \|T\|_2) / \text{sepi}$$

where sepi is the reciprocal condition number of x_i . You can compute the condition number sepi by calling `?trsna`.

See Also

Matrix Storage Schemes

`?trsna`

Estimates condition numbers for specified eigenvalues and right eigenvectors of an upper (quasi-) triangular matrix.

Syntax

```
lapack_int LAPACKE_strsna( int matrix_layout, char job, char howmny, const
lapack_logical* select, lapack_int n, const float* t, lapack_int ldt, const float* vl,
lapack_int ldvl, const float* vr, lapack_int ldvr, float* s, float* sep, lapack_int mm,
lapack_int* m );
```

```
lapack_int LAPACKE_dtrsna( int matrix_layout, char job, char howmny, const
lapack_logical* select, lapack_int n, const double* t, lapack_int ldt, const double*
vl, lapack_int ldvl, const double* vr, lapack_int ldvr, double* s, double* sep,
lapack_int mm, lapack_int* m );
```

```
lapack_int LAPACKE_ctrsna( int matrix_layout, char job, char howmny, const
lapack_logical* select, lapack_int n, const lapack_complex_float* t, lapack_int ldt,
const lapack_complex_float* vl, lapack_int ldvl, const lapack_complex_float* vr,
lapack_int ldvr, float* s, float* sep, lapack_int mm, lapack_int* m );
```

```
lapack_int LAPACKE_ztrsna( int matrix_layout, char job, char howmny, const
lapack_logical* select, lapack_int n, const lapack_complex_double* t, lapack_int ldt,
const lapack_complex_double* vl, lapack_int ldvl, const lapack_complex_double* vr,
lapack_int ldvr, double* s, double* sep, lapack_int mm, lapack_int* m );
```

Include Files

- `mk1.h`

Description

The routine estimates condition numbers for specified eigenvalues and/or right eigenvectors of an upper triangular matrix T (or, for real flavors, upper quasi-triangular matrix T in canonical Schur form). These are the same as the condition numbers of the eigenvalues and right eigenvectors of an original matrix $A = Z^* T^* Z^H$ (with unitary or, for real flavors, orthogonal Z), from which T may have been derived.

The routine computes the reciprocal of the condition number of an eigenvalue λ_i as $s_i = |v^{T*}u| / (||u||_E ||v||_E)$ for real flavors and $s_i = |v^{H*}u| / (||u||_E ||v||_E)$ for complex flavors,

where:

- u and v are the right and left eigenvectors of T , respectively, corresponding to λ_i .
- v^T/v^H denote transpose/conjugate transpose of v , respectively.

This reciprocal condition number always lies between zero (ill-conditioned) and one (well-conditioned).

An approximate error estimate for a computed eigenvalue λ_i is then given by $\varepsilon^* ||T|| / s_i$, where ε is the *machine precision*.

To estimate the reciprocal of the condition number of the right eigenvector corresponding to λ_i , the routine first calls [trexc](#) to reorder the diagonal elements of matrix T so that λ_i is in the leading position:

$$T = Q \begin{bmatrix} \lambda_i & C^H \\ 0 & T_{22} \end{bmatrix} Q^H$$

The reciprocal condition number of the eigenvector is then estimated as sep_i , the smallest singular value of the matrix $T_{22} - \lambda_i I$.

An approximate error estimate for a computed right eigenvector u corresponding to λ_i is then given by $\varepsilon^* ||T|| / sep_i$.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>job</code>	Must be 'E' or 'V' or 'B'. If <code>job</code> = 'E', then condition numbers for eigenvalues only are computed. If <code>job</code> = 'V', then condition numbers for eigenvectors only are computed. If <code>job</code> = 'B', then condition numbers for both eigenvalues and eigenvectors are computed.
<code>howmny</code>	Must be 'A' or 'S'. If <code>howmny</code> = 'A', then the condition numbers for all eigenpairs are computed. If <code>howmny</code> = 'S', then condition numbers for selected eigenpairs (as specified by <code>select</code>) are computed.

<i>select</i>	<p>Array, size at least $\max(1, n)$ if <i>howmny</i> = 'S' and at least 1 otherwise.</p> <p>Specifies the eigenpairs for which condition numbers are to be computed if <i>howmny</i> = 'S'.</p> <p><i>For real flavors:</i></p> <p>To select condition numbers for the eigenpair corresponding to the real eigenvalue λ_j, <i>select</i>[<i>j</i>] must be set 1;</p> <p>to select condition numbers for the eigenpair corresponding to a complex conjugate pair of eigenvalues λ_j and λ_{j+1}, <i>select</i>[<i>j</i> - 1] and/or <i>select</i>[<i>j</i>] must be set 1</p> <p><i>For complex flavors</i></p> <p>To select condition numbers for the eigenpair corresponding to the eigenvalue λ_j, <i>select</i>[<i>j</i>] must be set 1 <i>select</i> is not referenced if <i>howmny</i> = 'A'.</p>
<i>n</i>	The order of the matrix <i>T</i> ($n \geq 0$).
<i>t, vl, vr</i>	<p>Arrays:</p> <p><i>t</i> (size $\max(1, ldt * n)$) contains the <i>n</i>-by-<i>n</i> matrix <i>T</i>.</p> <p><i>vl</i> (size $\max(1, ldvl * mm)$ for column major layout and $\max(1, ldvl * n)$ for row major layout)</p> <p>If <i>job</i> = 'E' or 'B', then <i>vl</i> must contain the left eigenvectors of <i>T</i> (or of any matrix $Q * T * Q^H$ with <i>Q</i> unitary or orthogonal) corresponding to the eigenpairs specified by <i>howmny</i> and <i>select</i>. The eigenvectors must be stored in consecutive columns of <i>vl</i>, as returned by trevc or hsein.</p> <p>The array <i>vl</i> is not referenced if <i>job</i> = 'V'.</p> <p><i>vr</i> (size $\max(1, ldvr * mm)$ for column major layout and $\max(1, ldvr * n)$ for row major layout)</p> <p>If <i>job</i> = 'E' or 'B', then <i>vr</i> must contain the right eigenvectors of <i>T</i> (or of any matrix $Q * T * Q^H$ with <i>Q</i> unitary or orthogonal) corresponding to the eigenpairs specified by <i>howmny</i> and <i>select</i>. The eigenvectors must be stored in consecutive columns of <i>vr</i>, as returned by trevc or hsein.</p> <p>The array <i>vr</i> is not referenced if <i>job</i> = 'V'.</p>
<i>ldt</i>	The leading dimension of <i>t</i> ; at least $\max(1, n)$.
<i>ldvl</i>	<p>The leading dimension of <i>vl</i>.</p> <p>If <i>job</i> = 'E' or 'B', $ldvl \geq \max(1, n)$ for column major layout and $ldvl \geq \max(1, mm)$ for row major layout .</p> <p>If <i>job</i> = 'V', $ldvl \geq 1$.</p>
<i>ldvr</i>	<p>The leading dimension of <i>vr</i>.</p> <p>If <i>job</i> = 'E' or 'B', $ldvr \geq \max(1, n)$ for column major layout and $ldvr \geq \max(1, mm)$ for row major layout .</p> <p>If <i>job</i> = 'R', $ldvr \geq 1$.</p>

mm

The number of elements in the arrays *s* and *sep*, and the number of columns in *vl* and *vr* (if used). Must be at least *m* (the precise number required).

If *howmny* = 'A', *mm* = *n*;

if *howmny* = 'S', for *real flavorsmm* is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues.

for *complex flavorsmm* is the number of selected eigenpairs (see *select*).
Constraint:

$$0 \leq mm \leq n.$$

Output Parameters

s

Array, size at least $\max(1, mm)$ if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.

Contains the reciprocal condition numbers of the selected eigenvalues if *job* = 'E' or 'B', stored in consecutive elements of the array. Thus *s*[*j* - 1], *sep*[*j* - 1] and the *j*-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the *j*th eigenpair unless all eigenpairs have been selected).

For *real flavors*: for a complex conjugate pair of eigenvalues, two consecutive elements of *s* are set to the same value. The array *s* is not referenced if *job* = 'V'.

sep

Array, size at least $\max(1, mm)$ if *job* = 'V' or 'B' and at least 1 if *job* = 'E'. Contains the estimated reciprocal condition numbers of the selected right eigenvectors if *job* = 'V' or 'B', stored in consecutive elements of the array.

For *real flavors*: for a complex eigenvector, two consecutive elements of *sep* are set to the same value; if the eigenvalues cannot be reordered to compute *sep*[*j* - 1], then *sep*[*j* - 1] is set to zero; this can only occur when the true value would be very small anyway. The array *sep* is not referenced if *job* = 'E'.

m

For *complex flavors*: the number of selected eigenpairs.

If *howmny* = 'A', *m* is set to *n*.

For *real flavors*: the number of elements of *s* and/or *sep* actually used to store the estimated condition numbers.

If *howmny* = 'A', *m* is set to *n*.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed values *sep_i* may overestimate the true value, but seldom by a factor of more than 3.

`?trexc`Reorders the Schur factorization of a general matrix.

Syntax

```

lapack_int LAPACKE_strexc( int matrix_layout, char compq, lapack_int n, float* t,
lapack_int ldt, float* q, lapack_int ldq, lapack_int* ifst, lapack_int* ilst );

lapack_int LAPACKE_dtrexc( int matrix_layout, char compq, lapack_int n, double* t,
lapack_int ldt, double* q, lapack_int ldq, lapack_int* ifst, lapack_int* ilst );

lapack_int LAPACKE_ctrexc( int matrix_layout, char compq, lapack_int n,
lapack_complex_float* t, lapack_int ldt, lapack_complex_float* q, lapack_int ldq,
lapack_int ifst, lapack_int ilst );

lapack_int LAPACKE_ztrexc( int matrix_layout, char compq, lapack_int n,
lapack_complex_double* t, lapack_int ldt, lapack_complex_double* q, lapack_int ldq,
lapack_int ifst, lapack_int ilst );

```

Include Files

- `mkl.h`

Description

The routine reorders the Schur factorization of a general matrix $A = Q^*T^*Q^H$, so that the diagonal element or block of T with row index *ifst* is moved to row *ilst*.

The reordered Schur form S is computed by an unitary (or, for real flavors, orthogonal) similarity transformation: $S = Z^H * T * Z$. Optionally the updated matrix P of Schur vectors is computed as $P = Q * Z$, giving $A = P * S * P^H$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>compq</i>	Must be 'V' or 'N'. If <i>compq</i> = 'V', then the Schur vectors (Q) are updated. If <i>compq</i> = 'N', then no Schur vectors are updated.
<i>n</i>	The order of the matrix T ($n \geq 0$).
<i>t, q</i>	Arrays: <i>t</i> (size $\max(1, ldt * n)$) contains the n -by- n matrix T . <i>q</i> (size $\max(1, ldq * n)$) If <i>compq</i> = 'V', then <i>q</i> must contain Q (Schur vectors). If <i>compq</i> = 'N', then <i>q</i> is not referenced.
<i>ldt</i>	The leading dimension of <i>t</i> ; at least $\max(1, n)$.
<i>ldq</i>	The leading dimension of <i>q</i> ; If <i>compq</i> = 'N', then $ldq \geq 1$. If <i>compq</i> = 'V', then $ldq \geq \max(1, n)$.
<i>ifst, ilst</i>	$1 \leq ifst \leq n$; $1 \leq ilst \leq n$.

Must specify the reordering of the diagonal elements (or blocks, which is possible for real flavors) of the matrix T . The element (or block) with row index $ifst$ is moved to row $ilst$ by a sequence of exchanges between adjacent elements (or blocks).

Output Parameters

t	Overwritten by the updated matrix S .
q	If $compq = 'V'$, q contains the updated matrix of Schur vectors.
$ifst, ilst$	Overwritten for real flavors only. If $ifst$ pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; $ilst$ always points to the first row of the block in its final position (which may differ from its input value by ± 1).

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Application Notes

The computed matrix S is exactly similar to a matrix $T+E$, where $\|E\|_2 = O(\epsilon) * \|T\|_2$, and ϵ is the machine precision.

Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real.

The approximate number of floating-point operations is

for real flavors:	$6n(ifst-ilst)$ if $compq = 'N'$; $12n(ifst-ilst)$ if $compq = 'V'$;
for complex flavors:	$20n(ifst-ilst)$ if $compq = 'N'$; $40n(ifst-ilst)$ if $compq = 'V'$.

?trsen

Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers for the selected cluster of eigenvalues and respective invariant subspace.

Syntax

```
lapack_int LAPACKC_strsen( int matrix_layout, char job, char compq, const
lapack_logical* select, lapack_int n, float* t, lapack_int ldt, float* q, lapack_int
ldq, float* wr, float* wi, lapack_int* m, float* s, float* sep );

lapack_int LAPACKC_dtrsen( int matrix_layout, char job, char compq, const
lapack_logical* select, lapack_int n, double* t, lapack_int ldt, double* q, lapack_int
ldq, double* wr, double* wi, lapack_int* m, double* s, double* sep );
```

```
lapack_int LAPACKE_ctrssen( int matrix_layout, char job, char compq, const
lapack_logical* select, lapack_int n, lapack_complex_float* t, lapack_int ldt,
lapack_complex_float* q, lapack_int ldq, lapack_complex_float* w, lapack_int* m, float*
s, float* sep );
```

```
lapack_int LAPACKE_ztrssen( int matrix_layout, char job, char compq, const
lapack_logical* select, lapack_int n, lapack_complex_double* t, lapack_int ldt,
lapack_complex_double* q, lapack_int ldq, lapack_complex_double* w, lapack_int* m,
double* s, double* sep );
```

Include Files

- mkl.h

Description

The routine reorders the Schur factorization of a general matrix $A = Q^*T^*Q^T$ (for real flavors) or $A = Q^*T^*Q^H$ (for complex flavors) so that a selected cluster of eigenvalues appears in the leading diagonal elements (or, for real flavors, diagonal blocks) of the Schur form. The reordered Schur form R is computed by a unitary (orthogonal) similarity transformation: $R = Z^H * T * Z$. Optionally the updated matrix P of Schur vectors is computed as $P = Q * Z$, giving $A = P * R * P^H$.

Let

$$R = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{13} \end{bmatrix}$$

where the selected eigenvalues are precisely the eigenvalues of the leading m -by- m submatrix T_{11} . Let P be correspondingly partitioned as $(Q_1 Q_2)$ where Q_1 consists of the first m columns of Q . Then $A^* Q_1 = Q_1^* T_{11}$, and so the m columns of Q_1 form an orthonormal basis for the invariant subspace corresponding to the selected cluster of eigenvalues.

Optionally the routine also computes estimates of the reciprocal condition numbers of the average of the cluster of eigenvalues and of the invariant subspace.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

job

Must be 'N' or 'E' or 'V' or 'B'.

If *job* = 'N', then no condition numbers are required.

If `job = 'E'`, then only the condition number for the cluster of eigenvalues is computed.

If `job = 'V'`, then only the condition number for the invariant subspace is computed.

If `job = 'B'`, then condition numbers for both the cluster and the invariant subspace are computed.

Must be `'V'` or `'N'`.

If `compq = 'V'`, then `Q` of the Schur vectors is updated.

If `compq = 'N'`, then no Schur vectors are updated.

Array, size at least $\max(1, n)$.

Specifies the eigenvalues in the selected cluster. To select an eigenvalue λ_j , `select[j]` must be 1

For real flavors: to select a complex conjugate pair of eigenvalues λ_j and λ_{j+1} (corresponding 2 by 2 diagonal block), `select[j - 1]` and/or `select[j]` must be 1; the complex conjugate λ_j and λ_{j+1} must be either both included in the cluster or both excluded.

The order of the matrix T ($n \geq 0$).

Arrays:

`t` (size $\max(1, ldt * n)$) The upper quasi-triangular n -by- n matrix T , in Schur canonical form.

`q` (size $\max(1, ldq * n)$)

If `compq = 'V'`, then `q` must contain the matrix Q of Schur vectors.

If `compq = 'N'`, then `q` is not referenced.

The leading dimension of `t`; at least $\max(1, n)$.

The leading dimension of `q`;

If `compq = 'N'`, then $ldq \geq 1$.

If `compq = 'V'`, then $ldq \geq \max(1, n)$.

Output Parameters

Overwritten by the reordered matrix R in Schur canonical form with the selected eigenvalues in the leading diagonal blocks.

If `compq = 'V'`, `q` contains the updated matrix of Schur vectors; the first m columns of the Q form an orthogonal basis for the specified invariant subspace.

Array, size at least $\max(1, n)$. The recorded eigenvalues of R . The eigenvalues are stored in the same order as on the diagonal of R .

wr, wi	Arrays, size at least $\max(1, n)$. Contain the real and imaginary parts, respectively, of the reordered eigenvalues of R . The eigenvalues are stored in the same order as on the diagonal of R . Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.
m	<p><i>For complex flavors:</i> the dimension of the specified invariant subspaces, which is the same as the number of selected eigenvalues (see <i>select</i>).</p> <p><i>For real flavors:</i> the dimension of the specified invariant subspace. The value of m is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues (see <i>select</i>).</p> <p>Constraint: $0 \leq m \leq n$.</p>
s	<p>If $job = 'E'$ or $'B'$, s is a lower bound on the reciprocal condition number of the average of the selected cluster of eigenvalues.</p> <p>If $m = 0$ or n, then $s = 1$.</p> <p><i>For real flavors:</i> if $info = 1$, then s is set to zero. s is not referenced if $job = 'N'$ or $'V'$.</p>
sep	<p>If $job = 'V'$ or $'B'$, sep is the estimated reciprocal condition number of the specified invariant subspace.</p> <p>If $m = 0$ or n, then $sep = T$.</p> <p><i>For real flavors:</i> if $info = 1$, then sep is set to zero.</p> <p>sep is not referenced if $job = 'N'$ or $'E'$.</p>

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = 1$, the reordering of T failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); T may have been partially reordered, and wr and wi contain the eigenvalues in the same order as in T ; s and sep (if requested) are set to zero.

Application Notes

The computed matrix R is exactly similar to a matrix $T+E$, where $\|E\|_2 = O(\epsilon) * \|T\|_2$, and ϵ is the machine precision. The computed s cannot underestimate the true reciprocal condition number by more than a factor of $(\min(m, n-m))_{1/2}$; sep may differ from the true value by $(m*n-m^2)_{1/2}$. The angle between the computed invariant subspace and the true subspace is $O(\epsilon) * \|A\|_2 / sep$. Note that if a 2-by-2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2-by-2 block to break into two 1-by-1 blocks, that is, for a pair of complex eigenvalues to become purely real.

?trsyl

Solves Sylvester equation for real quasi-triangular or complex triangular matrices.

Syntax

```
lapack_int LAPACKE_strsyl( int matrix_layout, char trana, char tranb, lapack_int isgn,
lapack_int m, lapack_int n, const float* a, lapack_int lda, const float* b, lapack_int
ldb, float* c, lapack_int ldc, float* scale );
```

```
lapack_int LAPACKE_dtrsyl( int matrix_layout, char trana, char tranb, lapack_int isgn,
lapack_int m, lapack_int n, const double* a, lapack_int lda, const double* b,
lapack_int ldb, double* c, lapack_int ldc, double* scale );
```

```
lapack_int LAPACKE_ctrsyl( int matrix_layout, char trana, char tranb, lapack_int isgn,
lapack_int m, lapack_int n, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* c, lapack_int ldc,
float* scale );
```

```
lapack_int LAPACKE_ztrsyl( int matrix_layout, char trana, char tranb, lapack_int isgn,
lapack_int m, lapack_int n, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* c, lapack_int ldc,
double* scale );
```

Include Files

- mkl.h

Description

The routine solves the Sylvester matrix equation $\text{op}(A) * X \pm X * \text{op}(B) = \alpha * C$, where $\text{op}(A) = A$ or A^H , and the matrices A and B are upper triangular (or, for real flavors, upper quasi-triangular in canonical Schur form); $\alpha \leq 1$ is a scale factor determined by the routine to avoid overflow in X ; A is m -by- m , B is n -by- n , and C and X are both m -by- n . The matrix X is obtained by a straightforward process of back substitution.

The equation has a unique solution if and only if $\alpha_i \pm \beta_i \neq 0$, where $\{\alpha_i\}$ and $\{\beta_i\}$ are the eigenvalues of A and B , respectively, and the sign (+ or -) is the same as that used in the equation to be solved.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>trana</i>	Must be 'N' or 'T' or 'C'. If <i>trana</i> = 'N', then $\text{op}(A) = A$. If <i>trana</i> = 'T', then $\text{op}(A) = A^T$ (real flavors only). If <i>trana</i> = 'C' then $\text{op}(A) = A^H$.
<i>tranb</i>	Must be 'N' or 'T' or 'C'. If <i>tranb</i> = 'N', then $\text{op}(B) = B$. If <i>tranb</i> = 'T', then $\text{op}(B) = B^T$ (real flavors only). If <i>tranb</i> = 'C', then $\text{op}(B) = B^H$.
<i>isgn</i>	Indicates the form of the Sylvester equation. If <i>isgn</i> = +1, $\text{op}(A) * X + X * \text{op}(B) = \alpha * C$. If <i>isgn</i> = -1, $\text{op}(A) * X - X * \text{op}(B) = \alpha * C$.

m	The order of A , and the number of rows in X and C ($m \geq 0$).
n	The order of B , and the number of columns in X and C ($n \geq 0$).
a, b, c	Arrays: a (size $\max(1, lda * m)$) contains the matrix A . b (size $\max(1, ldb * n)$) contains the matrix B . c (size $\max(1, ldc * n)$ for column major layout and $\max(1, ldc * m)$ for row major layout) contains the matrix C .
lda	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
ldb	The leading dimension of b ; at least $\max(1, n)$.
ldc	The leading dimension of c ; at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.

Output Parameters

c	Overwritten by the solution matrix X .
$scale$	The value of the scale factor α .

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, A and B have common or close eigenvalues; perturbed values were used to solve the equation.

Application Notes

Let X be the exact, Y the corresponding computed solution, and R the residual matrix: $R = C - (AY \pm YB)$. Then the residual is always small:

$$\|R\|_F = O(\epsilon) * (\|A\|_F + \|B\|_F) * \|Y\|_F.$$

However, Y is not necessarily the exact solution of a slightly perturbed equation; in other words, the solution is not backwards stable.

For the forward error, the following bound holds:

$$\|Y - X\|_F \leq \|R\|_F / \text{sep}(A, B)$$

but this may be a considerable overestimate. See [Golub96] for a definition of $\text{sep}(A, B)$.

The approximate number of floating-point operations for real flavors is $m * n * (m + n)$. For complex flavors it is 4 times greater.

Generalized Nonsymmetric Eigenvalue Problems: LAPACK Computational Routines

This topic describes LAPACK routines for solving generalized nonsymmetric eigenvalue problems, reordering the generalized Schur factorization of a pair of matrices, as well as performing a number of related computational tasks.

A *generalized nonsymmetric eigenvalue problem* is as follows: given a pair of nonsymmetric (or non-Hermitian) n -by- n matrices A and B , find the *generalized eigenvalues* λ and the corresponding *generalized eigenvectors* x and y that satisfy the equations

$Ax = \lambda Bx$ (right generalized eigenvectors x)

and

$y^H A = \lambda y^H B$ (left generalized eigenvectors y).

Table "Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems" lists LAPACK routines used to solve the generalized nonsymmetric eigenvalue problems and the generalized Sylvester equation.

Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems

Routine name	Operation performed
ggghrd	Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.
ggbal	Balances a pair of general real or complex matrices.
ggbak	Forms the right or left eigenvectors of a generalized eigenvalue problem.
ggghd3	Reduces a pair of matrices to generalized upper Hessenberg form.
hgeqz	Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T).
tgevc	Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices
tgexc	Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.
tgsen	Reorders the <i>generalized</i> Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).
tgsyl	Solves the generalized Sylvester equation.
tgsyl	Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

?ggghrd

Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.

Syntax

```
lapack_int LAPACKE_sgghrd (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, float* a, lapack_int lda, float* b, lapack_int ldb,
float* q, lapack_int ldq, float* z, lapack_int ldz);
```

```
lapack_int LAPACKE_dgghrd (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, double* a, lapack_int lda, double* b, lapack_int ldb,
double* q, lapack_int ldq, double* z, lapack_int ldz);
```

```
lapack_int LAPACKE_cgghrd (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, lapack_complex_float* a, lapack_int lda,
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* q, lapack_int ldq,
lapack_complex_float* z, lapack_int ldz);
```

```
lapack_int LAPACKE_zgghrd (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, lapack_complex_double* a, lapack_int lda,
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* q, lapack_int ldq,
lapack_complex_double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine reduces a pair of real/complex matrices (A, B) to generalized upper Hessenberg form using orthogonal/unitary transformations, where A is a general matrix and B is upper triangular. The form of the generalized eigenvalue problem is $A^*x = \lambda^*B^*x$, and B is typically made upper triangular by computing its QR factorization and moving the orthogonal matrix Q to the left side of the equation.

This routine simultaneously reduces A to a Hessenberg matrix H :

$$Q^H A Z = H$$

and transforms B to another upper triangular matrix T :

$$Q^H B Z = T$$

in order to reduce the problem to its standard form $H^*y = \lambda^*T^*y$, where $y = Z^H x$.

The orthogonal/unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q_1 and Z_1 , so that

$$Q_1^* A Z_1^H = (Q_1^* Q)^* H^* (Z_1^* Z)^H$$

$$Q_1^* B Z_1^H = (Q_1^* Q)^* T^* (Z_1^* Z)^H$$

If Q_1 is the orthogonal/unitary matrix from the QR factorization of B in the original equation $A^*x = \lambda^*B^*x$, then the routine `?ggghrd` reduces the original problem to generalized Hessenberg form.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>compq</i>	<p>Must be 'N', 'I', or 'V'.</p> <p>If <i>compq</i> = 'N', matrix Q is not computed.</p> <p>If <i>compq</i> = 'I', Q is initialized to the unit matrix, and the orthogonal/unitary matrix Q is returned;</p> <p>If <i>compq</i> = 'V', Q must contain an orthogonal/unitary matrix Q_1 on entry, and the product Q_1^*Q is returned.</p>
<i>compz</i>	<p>Must be 'N', 'I', or 'V'.</p> <p>If <i>compz</i> = 'N', matrix Z is not computed.</p> <p>If <i>compz</i> = 'I', Z is initialized to the unit matrix, and the orthogonal/unitary matrix Z is returned;</p> <p>If <i>compz</i> = 'V', Z must contain an orthogonal/unitary matrix Z_1 on entry, and the product Z_1^*Z is returned.</p>
<i>n</i>	The order of the matrices A and B ($n \geq 0$).

<i>ilo, ihi</i>	<p><i>ilo</i> and <i>ihi</i> mark the rows and columns of <i>A</i> which are to be reduced. It is assumed that <i>A</i> is already upper triangular in rows and columns 1:<i>ilo</i>-1 and <i>ihi</i>+1:<i>n</i>. Values of <i>ilo</i> and <i>ihi</i> are normally set by a previous call to ggbal; otherwise they should be set to 1 and <i>n</i> respectively.</p> <p>Constraint:</p> <p>If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;</p> <p>if $n = 0$, then $ilo = 1$ and $ihi = 0$.</p>
<i>a, b, q, z</i>	<p>Arrays:</p> <p><i>a</i> (size $\max(1, lda*n)$) contains the <i>n</i>-by-<i>n</i> general matrix <i>A</i>.</p> <p><i>b</i> (size $\max(1, ldb*n)$) contains the <i>n</i>-by-<i>n</i> upper triangular matrix <i>B</i>.</p> <p><i>q</i> (size $\max(1, ldq*n)$)</p> <p>If <i>compq</i> = 'N', then <i>q</i> is not referenced.</p> <p>If <i>compq</i> = 'V', then <i>q</i> must contain the orthogonal/unitary matrix Q_1, typically from the <i>QR</i> factorization of <i>B</i>.</p> <p><i>z</i> (size $\max(1, ldz*n)$)</p> <p>If <i>compz</i> = 'N', then <i>z</i> is not referenced.</p> <p>If <i>compz</i> = 'V', then <i>z</i> must contain the orthogonal/unitary matrix Z_1.</p>
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; at least $\max(1, n)$.
<i>ldq</i>	<p>The leading dimension of <i>q</i>;</p> <p>If <i>compq</i> = 'N', then $ldq \geq 1$.</p> <p>If <i>compq</i> = 'I' or 'V', then $ldq \geq \max(1, n)$.</p>
<i>ldz</i>	<p>The leading dimension of <i>z</i>;</p> <p>If <i>compz</i> = 'N', then $ldz \geq 1$.</p> <p>If <i>compz</i> = 'I' or 'V', then $ldz \geq \max(1, n)$.</p>

Output Parameters

<i>a</i>	On exit, the upper triangle and the first subdiagonal of <i>A</i> are overwritten with the upper Hessenberg matrix <i>H</i> , and the rest is set to zero.
<i>b</i>	On exit, overwritten by the upper triangular matrix $T = Q^H * B * Z$. The elements below the diagonal are set to zero.
<i>q</i>	<p>If <i>compq</i> = 'I', then <i>q</i> contains the orthogonal/unitary matrix Q ;</p> <p>If <i>compq</i> = 'V', then <i>q</i> is overwritten by the product $Q_1 * Q$.</p>
<i>z</i>	<p>If <i>compz</i> = 'I', then <i>z</i> contains the orthogonal/unitary matrix Z;</p> <p>If <i>compz</i> = 'V', then <i>z</i> is overwritten by the product $Z_1 * Z$.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?ggbal

Balances a pair of general real or complex matrices.

Syntax

```
lapack_int LAPACKE_sggbal( int matrix_layout, char job, lapack_int n, float* a,
lapack_int lda, float* b, lapack_int ldb, lapack_int* ilo, lapack_int* ihi, float*
lscale, float* rscale );
```

```
lapack_int LAPACKE_dggbal( int matrix_layout, char job, lapack_int n, double* a,
lapack_int lda, double* b, lapack_int ldb, lapack_int* ilo, lapack_int* ihi, double*
lscale, double* rscale );
```

```
lapack_int LAPACKE_cggbal( int matrix_layout, char job, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
lapack_int* ilo, lapack_int* ihi, float* lscale, float* rscale );
```

```
lapack_int LAPACKE_zggbal( int matrix_layout, char job, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_int* ilo, lapack_int* ihi, double* lscale, double* rscale );
```

Include Files

- mkl.h

Description

The routine balances a pair of general real/complex matrices (*A*,*B*). This involves, first, permuting *A* and *B* by similarity transformations to isolate eigenvalues in the first 1 to *ilo*-1 and last *ihi*+1 to *n* elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns *ilo* to *ihi* to make the rows and columns as close in norm as possible. Both steps are optional. Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem $A*x = \lambda*B*x$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>job</i>	Specifies the operations to be performed on <i>A</i> and <i>B</i> . Must be 'N' or 'P' or 'S' or 'B'. If <i>job</i> = 'N', then no operations are done; simply set <i>ilo</i> =1, <i>ihi</i> = <i>n</i> , <i>lscale</i> [<i>i</i>] =1.0 and <i>rscale</i> [<i>i</i>]=1.0 for <i>i</i> = 0, ..., <i>n</i> - 1. If <i>job</i> = 'P', then permute only. If <i>job</i> = 'S', then scale only. If <i>job</i> = 'B', then both permute and scale.
<i>n</i>	The order of the matrices <i>A</i> and <i>B</i> (<i>n</i> ≥ 0).

a, b	<p>Arrays:</p> <p>a (size $\max(1, lda*n)$) contains the matrix A.</p> <p>b (size $\max(1, ldb*n)$) contains the matrix B.</p> <p>If $job = 'N'$, a and b are not referenced.</p>
lda	The leading dimension of a ; at least $\max(1, n)$.
ldb	The leading dimension of b ; at least $\max(1, n)$.

Output Parameters

a, b	Overwritten by the balanced matrices A and B , respectively.
ilo, ihi	<p>ilo and ihi are set to integers such that on exit $A_{i,j} = 0$ and $B_{i,j} = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.</p> <p>If $job = 'N'$ or $'S'$, then $ilo = 1$ and $ihi = n$.</p>
$lscale, rscale$	<p>Arrays, size at least $\max(1, n)$.</p> <p>$lscale$ contains details of the permutations and scaling factors applied to the left side of A and B.</p> <p>If P_j is the index of the row interchanged with row j, and D_j is the scaling factor applied to row j, then</p> $lscale[j - 1] = P_j, \text{ for } j = 1, \dots, ilo-1$ $= D_j, \text{ for } j = ilo, \dots, ihi$ $= P_j, \text{ for } j = ihi+1, \dots, n.$ <p>$rscale$ contains details of the permutations and scaling factors applied to the right side of A and B.</p> <p>If P_j is the index of the column interchanged with column j, and D_j is the scaling factor applied to column j, then</p> $rscale[j - 1] = P_j, \text{ for } j = 1, \dots, ilo-1$ $= D_j, \text{ for } j = ilo, \dots, ihi$ $= P_j, \text{ for } j = ihi+1, \dots, n$ <p>The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

?ggbak

Forms the right or left eigenvectors of a generalized eigenvalue problem.

Syntax

```
lapack_int LAPACKE_sggbak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const float* lscale, const float* rscale, lapack_int m,
float* v, lapack_int ldv );
```

```
lapack_int LAPACKE_dggbak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const double* lscale, const double* rscale, lapack_int
m, double* v, lapack_int ldv );
```

```
lapack_int LAPACKE_cggbak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const float* lscale, const float* rscale, lapack_int m,
lapack_complex_float* v, lapack_int ldv );
```

```
lapack_int LAPACKE_zggbak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const double* lscale, const double* rscale, lapack_int
m, lapack_complex_double* v, lapack_int ldv );
```

Include Files

- mkl.h

Description

The routine forms the right or left eigenvectors of a real/complex generalized eigenvalue problem

$$A*x = \lambda*B*x$$

by backward transformation on the computed eigenvectors of the balanced pair of matrices output by [ggbal](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>job</i>	Specifies the type of backward transformation required. Must be 'N', 'P', 'S', or 'B'. If <i>job</i> = 'N', then no operations are done; return. If <i>job</i> = 'P', then do backward transformation for permutation only. If <i>job</i> = 'S', then do backward transformation for scaling only. If <i>job</i> = 'B', then do backward transformation for both permutation and scaling. This argument must be the same as the argument <i>job</i> supplied to ggbal .
<i>side</i>	Must be 'L' or 'R'. If <i>side</i> = 'L', then <i>v</i> contains left eigenvectors. If <i>side</i> = 'R', then <i>v</i> contains right eigenvectors.
<i>n</i>	The number of rows of the matrix <i>V</i> ($n \geq 0$).
<i>ilo, ihi</i>	The integers <i>ilo</i> and <i>ihi</i> determined by ggbal . Constraint: If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, then $ilo = 1$ and $ihi = 0$.

<i>lscale, rscale</i>	<p>Arrays, size at least $\max(1, n)$.</p> <p>The array <i>lscale</i> contains details of the permutations and/or scaling factors applied to the left side of <i>A</i> and <i>B</i>, as returned by <code>?gghbd3</code>.</p> <p>The array <i>rscale</i> contains details of the permutations and/or scaling factors applied to the right side of <i>A</i> and <i>B</i>, as returned by <code>?gghbd3</code>.</p>
<i>m</i>	<p>The number of columns of the matrix <i>V</i></p> <p>($m \geq 0$).</p>
<i>v</i>	<p>Array <i>v</i>(size $\max(1, ldv*m)$ for column major layout and $\max(1, ldv*n)$ for row major layout) . Contains the matrix of right or left eigenvectors to be transformed, as returned by <code>tgevc</code>.</p>
<i>ldv</i>	<p>The leading dimension of <i>v</i>; at least $\max(1, n)$ for column major layout and at least $\max(1, m)$ for row major layout .</p>

Output Parameters

<i>v</i>	Overwritten by the transformed eigenvectors
----------	---

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?gghd3

*Reduces a pair of matrices to generalized upper
Hessenberg form.*

Syntax

```
lapack_int LAPACKE_sgghd3 (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, float * a, lapack_int lda, float * b, lapack_int ldb,
float * q, lapack_int ldq, float * z, lapack_int ldz);

lapack_int LAPACKE_dgghd3 (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, double * a, lapack_int lda, double * b, lapack_int ldb,
double * q, lapack_int ldq, double * z, lapack_int ldz);

lapack_int LAPACKE_cgghd3 (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, lapack_complex_float * a, lapack_int lda,
lapack_complex_float * b, lapack_int ldb, lapack_complex_float * q, lapack_int ldq,
lapack_complex_float * z, lapack_int ldz);

lapack_int LAPACKE_zgghd3 (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, lapack_complex_double * a, lapack_int lda,
lapack_complex_double * b, lapack_int ldb, lapack_complex_double * q, lapack_int ldq,
lapack_complex_double * z, lapack_int ldz);
```

Include Files

- mkl.h

Description

`?gghd3` reduces a pair of real or complex matrices (A , B) to generalized upper Hessenberg form using orthogonal/unitary transformations, where A is a general matrix and B is upper triangular. The form of the generalized eigenvalue problem is

$$A*x = \lambda*B*x,$$

and B is typically made upper triangular by computing its QR factorization and moving the orthogonal/unitary matrix Q to the left side of the equation.

This subroutine simultaneously reduces A to a Hessenberg matrix H :

$$Q^T*A*Z = H \text{ for real flavors}$$

or

$$Q^T*A*Z = H \text{ for complex flavors}$$

and transforms B to another upper triangular matrix T :

$$Q^T*B*Z = T \text{ for real flavors}$$

or

$$Q^T*B*Z = T \text{ for complex flavors}$$

in order to reduce the problem to its standard form

$$H*y = \lambda*T*y$$

where $y = Z^T*x$ for real flavors

or

$$y = Z^T*x \text{ for complex flavors.}$$

The orthogonal/unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q_1 and Z_1 , so that

for real flavors:

$$Q_1 * A * Z_1^T = (Q_1 * Q) * H * (Z_1 * Z)^T$$

$$Q_1 * B * Z_1^T = (Q_1 * Q) * T * (Z_1 * Z)^T$$

for complex flavors:

$$Q_1 * A * Z_1^H = (Q_1 * Q) * H * (Z_1 * Z)^T$$

$$Q_1 * B * Z_1^T = (Q_1 * Q) * T * (Z_1 * Z)^T$$

If Q_1 is the orthogonal/unitary matrix from the QR factorization of B in the original equation $A*x = \lambda*B*x$, then `?gghd3` reduces the original problem to generalized Hessenberg form.

This is a blocked variant of `?gghrd`, using matrix-matrix multiplications for parts of the computation to enhance performance.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>compq</code>	<p>= 'N': do not compute q;</p> <p>= 'I': q is initialized to the unit matrix, and the orthogonal/unitary matrix Q is returned;</p>

	<p>= 'V': q must contain an orthogonal/unitary matrix Q_1 on entry, and the product $Q_1 * q$ is returned.</p>
<i>compz</i>	<p>= 'N': do not compute z;</p> <p>= 'I': z is initialized to the unit matrix, and the orthogonal/unitary matrix Z is returned;</p> <p>= 'V': z must contain an orthogonal/unitary matrix Z_1 on entry, and the product $Z_1 * z$ is returned.</p>
<i>n</i>	<p>The order of the matrices A and B.</p> <p>$n \geq 0$.</p>
<i>ilo, ihi</i>	<p>ilo and ihi mark the rows and columns of a which are to be reduced. It is assumed that a is already upper triangular in rows and columns $1:ilo - 1$ and $ihi + 1:n$. ilo and ihi are normally set by a previous call to ?ggbal; otherwise they should be set to 1 and n, respectively.</p> <p>$1 \leq ilo \leq ihi \leq n$, if $n > 0$; $ilo=1$ and $ihi=0$, if $n=0$.</p>
<i>a</i>	<p>Array, size ($lda * n$).</p> <p>On entry, the n-by-n general matrix to be reduced.</p>
<i>lda</i>	<p>The leading dimension of the array a.</p> <p>$lda \geq \max(1, n)$.</p>
<i>b</i>	<p>Array, ($ldb * n$).</p> <p>On entry, the n-by-n upper triangular matrix B.</p>
<i>ldb</i>	<p>The leading dimension of the array b.</p> <p>$ldb \geq \max(1, n)$.</p>
<i>q</i>	<p>Array, size ($ldq * n$).</p> <p>On entry, if <i>compq</i> = 'V', the orthogonal/unitary matrix Q_1, typically from the QR factorization of b.</p>
<i>ldq</i>	<p>The leading dimension of the array q.</p> <p>$ldq \geq n$ if <i>compq</i>='V' or 'I'; $ldq \geq 1$ otherwise.</p>
<i>z</i>	<p>Array, size ($ldz * n$).</p> <p>On entry, if <i>compz</i> = 'V', the orthogonal/unitary matrix Z_1.</p> <p>Not referenced if <i>compz</i>='N'.</p>
<i>ldz</i>	<p>The leading dimension of the array z. $ldz \geq n$ if <i>compz</i>='V' or 'I'; $ldz \geq 1$ otherwise.</p>

Output Parameters

<i>a</i>	On exit, the upper triangle and the first subdiagonal of a are overwritten with the upper Hessenberg matrix H , and the rest is set to zero.
----------	--

<i>b</i>	On exit, the upper triangular matrix $T = Q^T B Z$ for real flavors or $T = Q^H B Z$ for complex flavors. The elements below the diagonal are set to zero.
<i>q</i>	On exit, if <i>compq</i> ='I', the orthogonal/unitary matrix <i>Q</i> , and if <i>compq</i> ='V', the product $Q_1^* Q$. Not referenced if <i>compq</i> ='N'.
<i>z</i>	On exit, if <i>compz</i> ='I', the orthogonal/unitary matrix <i>Z</i> , and if <i>compz</i> ='V', the product $Z_1^* Z$. Not referenced if <i>compz</i> ='N'.

Return Values

This function returns a value *info*.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

Application Notes

This routine reduces *A* to Hessenberg form and maintains *B* in using a blocked variant of Moler and Stewart's original algorithm, as described by Kagstrom, Kressner, Quintana-Orti, and Quintana-Orti (BIT 2008).

?hgeqz

Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H,T).

Syntax

```
lapack_int LAPACKE_shgeqz( int matrix_layout, char job, char compq, char compz,
lapack_int n, lapack_int ilo, lapack_int ihi, float* h, lapack_int ldh, float* t,
lapack_int ldt, float* alphas, float* alphai, float* beta, float* q, lapack_int ldq,
float* z, lapack_int ldz );

lapack_int LAPACKE_dhgeqz( int matrix_layout, char job, char compq, char compz,
lapack_int n, lapack_int ilo, lapack_int ihi, double* h, lapack_int ldh, double* t,
lapack_int ldt, double* alphas, double* alphai, double* beta, double* q, lapack_int ldq,
double* z, lapack_int ldz );

lapack_int LAPACKE_chgeqz( int matrix_layout, char job, char compq, char compz,
lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_float* h, lapack_int ldh,
lapack_complex_float* t, lapack_int ldt, lapack_complex_float* alpha,
lapack_complex_float* beta, lapack_complex_float* q, lapack_int ldq,
lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhgeqz( int matrix_layout, char job, char compq, char compz,
lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_double* h, lapack_int ldh,
lapack_complex_double* t, lapack_int ldt, lapack_complex_double* alpha,
lapack_complex_double* beta, lapack_complex_double* q, lapack_int ldq,
lapack_complex_double* z, lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes the eigenvalues of a real/complex matrix pair (H,T) , where H is an upper Hessenberg matrix and T is upper triangular, using the double-shift version (for real flavors) or single-shift version (for complex flavors) of the QZ method. Matrix pairs of this type are produced by the reduction to generalized upper Hessenberg form of a real/complex matrix pair (A,B) :

$$A = Q_1^* H^* Z_1^H, B = Q_1^* T^* Z_1^H,$$

as computed by `?gghrd`.

For real flavors:

If `job = 'S'`, then the Hessenberg-triangular pair (H,T) is reduced to generalized Schur form,

$$H = Q^* S^* Z^T, T = Q^* P^* Z^T,$$

where Q and Z are orthogonal matrices, P is an upper triangular matrix, and S is a quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks. The 1-by-1 blocks correspond to real eigenvalues of the matrix pair (H,T) and the 2-by-2 blocks correspond to complex conjugate pairs of eigenvalues.

Additionally, the 2-by-2 upper triangular diagonal blocks of P corresponding to 2-by-2 blocks of S are reduced to positive diagonal form, that is, if $S_{j+1,j}$ is non-zero, then $P_{j+1,j} = P_{j,j+1} = 0$, $P_{j,j} > 0$, and $P_{j+1,j+1} > 0$.

For complex flavors:

If `job = 'S'`, then the Hessenberg-triangular pair (H,T) is reduced to generalized Schur form,

$$H = Q^* S^* Z^H, T = Q^* P^* Z^H,$$

where Q and Z are unitary matrices, and S and P are upper triangular.

For all function flavors:

Optionally, the orthogonal/unitary matrix Q from the generalized Schur factorization may be post-multiplied by an input matrix Q_1 , and the orthogonal/unitary matrix Z may be post-multiplied by an input matrix Z_1 .

If Q_1 and Z_1 are the orthogonal/unitary matrices from `?gghrd` that reduced the matrix pair (A,B) to generalized upper Hessenberg form, then the output matrices $Q_1 Q$ and $Z_1 Z$ are the orthogonal/unitary factors from the generalized Schur factorization of (A,B) :

$$A = (Q_1 Q)^* S^* (Z_1 Z)^H, B = (Q_1 Q)^* P^* (Z_1 Z)^H.$$

To avoid overflow, eigenvalues of the matrix pair (H,T) (equivalently, of (A,B)) are computed as a pair of values (α, β) . For `chgeqz/zhgeqz`, α and β are complex, and for `shgeqz/dhgeqz`, α is complex and β real. If β is nonzero, $\lambda = \alpha/\beta$ is an eigenvalue of the generalized nonsymmetric eigenvalue problem (GNEP)

$$A^* x = \lambda^* B^* x$$

and if α is nonzero, $\mu = \beta/\alpha$ is an eigenvalue of the alternate form of the GNEP

$$\mu^* A^* y = B^* y.$$

Real eigenvalues (for real flavors) or the values of α and β for the i -th eigenvalue (for complex flavors) can be read directly from the generalized Schur form:

$$\alpha = S_{i,i}, \beta = P_{i,i}.$$

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>job</code>	Specifies the operations to be performed. Must be 'E' or 'S'.

	<p>If <code>job = 'E'</code>, then compute eigenvalues only;</p> <p>If <code>job = 'S'</code>, then compute eigenvalues and the Schur form.</p>
<code>compq</code>	<p>Must be <code>'N'</code>, <code>'I'</code>, or <code>'V'</code>.</p> <p>If <code>compq = 'N'</code>, left Schur vectors (q) are not computed;</p> <p>If <code>compq = 'I'</code>, q is initialized to the unit matrix and the matrix of left Schur vectors of (H,T) is returned;</p> <p>If <code>compq = 'V'</code>, q must contain an orthogonal/unitary matrix Q_1 on entry and the product Q_1*Q is returned.</p>
<code>compz</code>	<p>Must be <code>'N'</code>, <code>'I'</code>, or <code>'V'</code>.</p> <p>If <code>compz = 'N'</code>, right Schur vectors (z) are not computed;</p> <p>If <code>compz = 'I'</code>, z is initialized to the unit matrix and the matrix of right Schur vectors of (H,T) is returned;</p> <p>If <code>compz = 'V'</code>, z must contain an orthogonal/unitary matrix Z_1 on entry and the product Z_1*Z is returned.</p>
<code>n</code>	<p>The order of the matrices H, T, Q, and Z</p> <p>($n \geq 0$).</p>
<code>ilo, ihi</code>	<p>ilo and ihi mark the rows and columns of H which are in Hessenberg form. It is assumed that H is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$.</p> <p>Constraint:</p> <p>If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;</p> <p>if $n = 0$, then $ilo = 1$ and $ihi = 0$.</p>
<code>h, t, q, z</code>	<p>Arrays:</p> <p>On entry, h (size $\max(1, ldh*n)$) contains the n-by-n upper Hessenberg matrix H.</p> <p>On entry, t (size $\max(1, ldt*n)$) contains the n-by-n upper triangular matrix T.</p> <p>q (size $\max(1, ldq*n)$) :</p> <p>On entry, if <code>compq = 'V'</code>, this array contains the orthogonal/unitary matrix Q_1 used in the reduction of (A,B) to generalized Hessenberg form.</p> <p>If <code>compq = 'N'</code>, then q is not referenced.</p> <p>z (size $\max(1, ldz*n)$) :</p> <p>On entry, if <code>compz = 'V'</code>, this array contains the orthogonal/unitary matrix Z_1 used in the reduction of (A,B) to generalized Hessenberg form.</p> <p>If <code>compz = 'N'</code>, then z is not referenced.</p>
<code>ldh</code>	The leading dimension of h ; at least $\max(1, n)$.
<code>ldt</code>	The leading dimension of t ; at least $\max(1, n)$.
<code>ldq</code>	The leading dimension of q ;

If `compq = 'N'`, then $ldq \geq 1$.

If `compq = 'I' or 'V'`, then $ldq \geq \max(1, n)$.

`ldz`

The leading dimension of `z`;

If `compq = 'N'`, then $ldz \geq 1$.

If `compq = 'I' or 'V'`, then $ldz \geq \max(1, n)$.

Output Parameters

`h`

For real flavors:

If `job = 'S'`, then on exit `h` contains the upper quasi-triangular matrix `S` from the generalized Schur factorization.

If `job = 'E'`, then on exit the diagonal blocks of `h` match those of `S`, but the rest of `h` is unspecified.

For complex flavors:

If `job = 'S'`, then, on exit, `h` contains the upper triangular matrix `S` from the generalized Schur factorization.

If `job = 'E'`, then on exit the diagonal of `h` matches that of `S`, but the rest of `h` is unspecified.

`t`

If `job = 'S'`, then, on exit, `t` contains the upper triangular matrix `P` from the generalized Schur factorization.

For real flavors:

2-by-2 diagonal blocks of `P` corresponding to 2-by-2 blocks of `S` are reduced to positive diagonal form, that is, if $h(j+1,j)$ is non-zero, then $t(j+1,j) = t(j,j+1) = 0$ and $t(j,j)$ and $t(j+1,j+1)$ will be positive.

If `job = 'E'`, then on exit the diagonal blocks of `t` match those of `P`, but the rest of `t` is unspecified.

For complex flavors:

if `job = 'E'`, then on exit the diagonal of `t` matches that of `P`, but the rest of `t` is unspecified.

`alphar, alphai`

Arrays, size at least $\max(1, n)$. The real and imaginary parts, respectively, of each scalar `alpha` defining an eigenvalue of GNEP.

If `alphai[j - 1]` is zero, then the j -th eigenvalue is real; if positive, then the j -th and $(j+1)$ -th eigenvalues are a complex conjugate pair, with

$$\text{alphai}[j] = -\text{alphai}[j - 1].$$

`alpha`

Array, size at least $\max(1, n)$.

The complex scalars `alpha` that define the eigenvalues of GNEP. $\text{alphai}[i - 1] = S_i$, i in the generalized Schur factorization.

`beta`

Array, size at least $\max(1, n)$.

For real flavors:

The scalars `beta` that define the eigenvalues of GNEP.

Together, the quantities $\alpha = (\alpha_{\text{real}}[j - 1], \alpha_{\text{imag}}[j - 1])$ and $\beta = \beta[j - 1]$ represent the j -th eigenvalue of the matrix pair (A, B) , in one of the forms $\lambda = \alpha/\beta$ or $\mu = \beta/\alpha$. Since either λ or μ may overflow, they should not, in general, be computed.

For complex flavors:

The real non-negative scalars β that define the eigenvalues of GNEP.

$\beta[i - 1] = P_{i, i}$ in the generalized Schur factorization. Together, the quantities $\alpha = \alpha[j - 1]$ and $\beta = \beta[j - 1]$ represent the j -th eigenvalue of the matrix pair (A, B) , in one of the forms $\lambda = \alpha/\beta$ or $\mu = \beta/\alpha$. Since either λ or μ may overflow, they should not, in general, be computed.

q

On exit, if $\text{compq} = 'I'$, q is overwritten by the orthogonal/unitary matrix of left Schur vectors of the pair (H, T) , and if $\text{compq} = 'V'$, q is overwritten by the orthogonal/unitary matrix of left Schur vectors of (A, B) .

z

On exit, if $\text{compz} = 'I'$, z is overwritten by the orthogonal/unitary matrix of right Schur vectors of the pair (H, T) , and if $\text{compz} = 'V'$, z is overwritten by the orthogonal/unitary matrix of right Schur vectors of (A, B) .

Return Values

This function returns a value info .

If $\text{info}=0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value.

If $\text{info} = 1, \dots, n$, the QZ iteration did not converge.

(H, T) is not in Schur form, but $\alpha_{\text{real}}[i - 1]$, $\alpha_{\text{imag}}[i - 1]$ (for real flavors), $\alpha[i - 1]$ (for complex flavors), and $\beta[i - 1]$, $i=\text{info}+1, \dots, n$ should be correct.

If $\text{info} = n+1, \dots, 2n$, the shift calculation failed.

(H, T) is not in Schur form, but $\alpha_{\text{real}}[i - 1]$, $\alpha_{\text{imag}}[i - 1]$ (for real flavors), $\alpha[i - 1]$ (for complex flavors), and $\beta[i - 1]$, $i = \text{info}-n+1, \dots, n$ should be correct.

?tgevc

Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices.

Syntax

```
lapack_int LAPACKE_stgevc (int matrix_layout, char side, char howmny, const
lapack_logical* select, lapack_int n, const float* s, lapack_int lds, const float* p,
lapack_int ldp, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr, lapack_int mm,
lapack_int* m);
```

```
lapack_int LAPACKE_dtgevc (int matrix_layout, char side, char howmny, const
lapack_logical* select, lapack_int n, const double* s, lapack_int lds, const double* p,
lapack_int ldp, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr, lapack_int mm,
lapack_int* m);
```

```
lapack_int LAPACKE_ctgevc (int matrix_layout, char side, char howmny, const
lapack_logical* select, lapack_int n, const lapack_complex_float* s, lapack_int lds,
const lapack_complex_float* p, lapack_int ldp, lapack_complex_float* vl, lapack_int
ldvl, lapack_complex_float* vr, lapack_int ldvr, lapack_int mm, lapack_int* m);

lapack_int LAPACKE_ztgevc (int matrix_layout, char side, char howmny, const
lapack_logical* select, lapack_int n, const lapack_complex_double* s, lapack_int lds,
const lapack_complex_double* p, lapack_int ldp, lapack_complex_double* vl, lapack_int
ldvl, lapack_complex_double* vr, lapack_int ldvr, lapack_int mm, lapack_int* m);
```

Include Files

- mkl.h

Description

The routine computes some or all of the right and/or left eigenvectors of a pair of real/complex matrices (S,P) , where S is quasi-triangular (for real flavors) or upper triangular (for complex flavors) and P is upper triangular.

Matrix pairs of this type are produced by the generalized Schur factorization of a real/complex matrix pair (A,B) :

$$A = Q^* S^* Z^H, B = Q^* P^* Z^H$$

as computed by ?gghrd plus ?hgeqz.

The right eigenvector x and the left eigenvector y of (S,P) corresponding to an eigenvalue w are defined by:

$$S^* x = w^* P^* x, y^H S = w^* y^H P$$

The eigenvalues are not input to this routine, but are computed directly from the diagonal blocks or diagonal elements of S and P .

This routine returns the matrices X and/or Y of right and left eigenvectors of (S,P) , or the products $Z^* X$ and/or $Q^* Y$, where Z and Q are input matrices.

If Q and Z are the orthogonal/unitary factors from the generalized Schur factorization of a matrix pair (A,B) , then $Z^* X$ and $Q^* Y$ are the matrices of right and left eigenvectors of (A,B) .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	Must be 'R', 'L', or 'B'. If <i>side</i> = 'R', compute right eigenvectors only. If <i>side</i> = 'L', compute left eigenvectors only. If <i>side</i> = 'B', compute both right and left eigenvectors.
<i>howmny</i>	Must be 'A', 'B', or 'S'. If <i>howmny</i> = 'A', compute all right and/or left eigenvectors. If <i>howmny</i> = 'B', compute all right and/or left eigenvectors, backtransformed by the matrices in <i>vr</i> and/or <i>vl</i> . If <i>howmny</i> = 'S', compute selected right and/or left eigenvectors, specified by the logical array <i>select</i> .

select

Array, size at least $\max(1, n)$.

If *howmny* = 'S', *select* specifies the eigenvectors to be computed.

If *howmny* = 'A' or 'B', *select* is not referenced.

For real flavors:

If $w[j]$ is a real eigenvalue, the corresponding real eigenvector is computed if *select*[*j*] is 1.

If $w[j]$ and $\omega[j + 1]$ are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either *select*[*j*] or *select*[*j* + 1] is 1, and on exit *select*[*j*] is set to 1 and *select*[*j* + 1] is set to 0.

For complex flavors:

The eigenvector corresponding to the *j*-th eigenvalue is computed if *select*[*j*] is 1.

n

The order of the matrices *S* and *P* ($n \geq 0$).

s, p, vl, vr

Arrays:

s (size $\max(1, lds * n)$) contains the matrix *S* from a generalized Schur factorization as computed by ?hgeqz. This matrix is upper quasi-triangular for real flavors, and upper triangular for complex flavors.

p (size $\max(1, ldp * n)$) contains the upper triangular matrix *P* from a generalized Schur factorization as computed by ?hgeqz.

For real flavors, 2-by-2 diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* must be in positive diagonal form.

For complex flavors, *P* must have real diagonal elements.

If *side* = 'L' or 'B' and *howmny* = 'B', *vl* (size $\max(1, ldvl * mm)$ for column major layout and $\max(1, ldvl * n)$ for row major layout) must contain an *n*-by-*n* matrix *Q* (usually the orthogonal/unitary matrix *Q* of left Schur vectors returned by ?hgeqz).

If *side* = 'R', *vl* is not referenced.

If *side* = 'R' or 'B' and *howmny* = 'B', *vr* (size $\max(1, ldvr * mm)$ for column major layout and $\max(1, ldvr * n)$ for row major layout) must contain an *n*-by-*n* matrix *Z* (usually the orthogonal/unitary matrix *Z* of right Schur vectors returned by ?hgeqz).

If *side* = 'L', *vr* is not referenced.

lds

The leading dimension of *s*; at least $\max(1, n)$.

ldp

The leading dimension of *p*; at least $\max(1, n)$.

ldvl

The leading dimension of *vl*;

If *side* = 'L' or 'B', then $ldvl \geq n$ for column major layout and $ldvl \geq \max(1, mm)$ for row major layout.

If *side* = 'R', then $ldvl \geq 1$.

ldvr

The leading dimension of *vr*;

If $side = 'R'$ or $'B'$, then $ldvr \geq n$ for column major layout and $ldvr \geq \max(1, mm)$ for row major layout.

If $side = 'L'$, then $ldvr \geq 1$.

mm

The number of columns in the arrays vl and/or vr ($mm \geq m$).

Output Parameters

vl

On exit, if $side = 'L'$ or $'B'$, vl contains:

if $howmny = 'A'$, the matrix Y of left eigenvectors of (S,P) ;

if $howmny = 'B'$, the matrix $Q*Y$;

if $howmny = 'S'$, the left eigenvectors of (S,P) specified by $select$, stored consecutively in the columns of vl , in the same order as their eigenvalues.

For real flavors:

A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

vr

On exit, if $side = 'R'$ or $'B'$, vr contains:

if $howmny = 'A'$, the matrix X of right eigenvectors of (S,P) ;

if $howmny = 'B'$, the matrix $Z*X$;

if $howmny = 'S'$, the right eigenvectors of (S,P) specified by $select$, stored consecutively in the columns of vr , in the same order as their eigenvalues.

For real flavors:

A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

m

The number of columns in the arrays vl and/or vr actually used to store the eigenvectors.

If $howmny = 'A'$ or $'B'$, m is set to n .

For real flavors:

Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.

For complex flavors:

Each selected eigenvector occupies one column.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

For real flavors:

if $info = i > 0$, the 2-by-2 block $(i:i+1)$ does not have a complex eigenvalue.

?tgexc

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.

Syntax

```
lapack_int LAPACKE_stgexc (int matrix_layout, lapack_logical wantq, lapack_logical wantz, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float* q, lapack_int ldq, float* z, lapack_int ldz, lapack_int* ifst, lapack_int* ilst);

lapack_int LAPACKE_dtgexc (int matrix_layout, lapack_logical wantq, lapack_logical wantz, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double* q, lapack_int ldq, double* z, lapack_int ldz, lapack_int* ifst, lapack_int* ilst);

lapack_int LAPACKE_ctgexc (int matrix_layout, lapack_logical wantq, lapack_logical wantz, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* q, lapack_int ldq, lapack_complex_float* z, lapack_int ldz, lapack_int ifst, lapack_int ilst);

lapack_int LAPACKE_ztgexc (int matrix_layout, lapack_logical wantq, lapack_logical wantz, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* q, lapack_int ldq, lapack_complex_double* z, lapack_int ldz, lapack_int ifst, lapack_int ilst);
```

Include Files

- mkl.h

Description

The routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (A,B) using an orthogonal/unitary equivalence transformation

$$(A, B) = Q^* (A, B) * Z^H,$$

so that the diagonal block of (A, B) with row index *ifst* is moved to row *ilst*. Matrix pair (A, B) must be in a generalized real-Schur/Schur canonical form (as returned by [gges](#)), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks and B is upper triangular. Optionally, the matrices Q and Z of generalized Schur vectors are updated.

$$Q_{in} * A_{in} * Z_{in}^T = Q_{out} * A_{out} * Z_{out}^T$$

$$Q_{in} * B_{in} * Z_{in}^T = Q_{out} * B_{out} * Z_{out}^T.$$

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>wantq, wantz</i>	<p>If <i>wantq</i> = 1, update the left transformation matrix Q;</p> <p>If <i>wantq</i> = 0, do not update Q;</p> <p>If <i>wantz</i> = 1, update the right transformation matrix Z;</p> <p>If <i>wantz</i> = 0, do not update Z.</p>
<i>n</i>	The order of the matrices A and B ($n \geq 0$).

a, b, q, z	<p>Arrays:</p> <p>a (size $\max(1, lda*n)$) contains the matrix A.</p> <p>b (size $\max(1, ldb*n)$) contains the matrix B.</p> <p>q (size at least 1 if $wantq = 0$ and at least $\max(1, ldq*n)$ if $wantq = 1$)</p> <p>If $wantq = 0$, then q is not referenced.</p> <p>If $wantq = 1$, then q must contain the orthogonal/unitary matrix Q.</p> <p>z (size at least 1 if $wantz = 0$ and at least $\max(1, ldz*n)$ if $wantz = 1$)</p> <p>If $wantz = 0$, then z is not referenced.</p> <p>If $wantz = 1$, then z must contain the orthogonal/unitary matrix Z.</p>
lda	The leading dimension of a ; at least $\max(1, n)$.
ldb	The leading dimension of b ; at least $\max(1, n)$.
ldq	<p>The leading dimension of q;</p> <p>If $wantq = 0$, then $ldq \geq 1$.</p> <p>If $wantq = 1$, then $ldq \geq \max(1, n)$.</p>
ldz	<p>The leading dimension of z;</p> <p>If $wantz = 0$, then $ldz \geq 1$.</p> <p>If $wantz = 1$, then $ldz \geq \max(1, n)$.</p>
$ifst, ilst$	Specify the reordering of the diagonal blocks of (A, B) . The block with row index $ifst$ is moved to row $ilst$, by a sequence of swapping between adjacent blocks. Constraint: $1 \leq ifst, ilst \leq n$.

Output Parameters

a, b, q, z	Overwritten by the updated matrices A, B, Q , and Z respectively.
$ifst, ilst$	<p>Overwritten for real flavors only.</p> <p>If $ifst$ pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; $ilst$ always points to the first row of the block in its final position (which may differ from its input value by ± 1).</p>

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = 1$, the transformed matrix pair (A, B) would be too far from generalized Schur form; the problem is ill-conditioned. (A, B) may have been partially reordered, and $ilst$ points to the first row of the current position of the block being moved.

?tgsen

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B) .

Syntax

```
lapack_int LAPACKE_stgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n, float* a, lapack_int
lda, float* b, lapack_int ldb, float* alphas, float* alphas_i, float* beta, float* q,
lapack_int ldq, float* z, lapack_int ldz, lapack_int* m, float* pl, float* pr, float*
dif );

lapack_int LAPACKE_dtgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n, double* a, lapack_int
lda, double* b, lapack_int ldb, double* alphas, double* alphas_i, double* beta, double* q,
lapack_int ldq, double* z, lapack_int ldz, lapack_int* m, double* pl, double* pr,
double* dif );

lapack_int LAPACKE_ctgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n, lapack_complex_float*
a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, lapack_complex_float*
alpha, lapack_complex_float* beta, lapack_complex_float* q, lapack_int ldq,
lapack_complex_float* z, lapack_int ldz, lapack_int* m, float* pl, float* pr, float*
dif );

lapack_int LAPACKE_ztgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* alpha, lapack_complex_double* beta, lapack_complex_double* q,
lapack_int ldq, lapack_complex_double* z, lapack_int ldz, lapack_int* m, double* pl,
double* pr, double* dif );
```

Include Files

- mkl.h

Description

The routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (A, B) (in terms of an orthogonal/unitary equivalence transformation $Q^T(A, B)Z$ for real flavors or $Q^H(A, B)Z$ for complex flavors), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (A, B) . The leading columns of Q and Z form orthonormal/unitary bases of the corresponding left and right eigenspaces (deflating subspaces).

(A, B) must be in generalized real-Schur/Schur canonical form (as returned by [gges](#)), that is, A and B are both upper triangular.

?tgsen also computes the generalized eigenvalues

$$\omega_j = (\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j) \text{ (for real flavors)}$$

$$\omega_j = \text{alpha}(j)/\text{beta}(j) \text{ (for complex flavors)}$$

of the reordered matrix pair (A, B) .

Optionally, the routine computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are $\text{Difu}[(A_{11}, B_{11}), (A_{22}, B_{22})]$ and $\text{Difl}[(A_{11}, B_{11}), (A_{22}, B_{22})]$, that is, the separation(s) between the matrix pairs (A_{11}, B_{11}) and (A_{22}, B_{22}) that correspond to the selected cluster and the eigenvalues outside the cluster, respectively, and norms of "projections" onto left and right eigenspaces with respect to the selected cluster in the (1,1)-block.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>ijob</i>	<p>Specifies whether condition numbers are required for the cluster of eigenvalues (<i>pl</i> and <i>pr</i>) or the deflating subspaces <i>Difu</i> and <i>Difl</i>.</p> <p>If <i>ijob</i> = 0, only reorder with respect to <i>select</i>;</p> <p>If <i>ijob</i> = 1, reciprocal of norms of "projections" onto left and right eigenspaces with respect to the selected cluster (<i>pl</i> and <i>pr</i>);</p> <p>If <i>ijob</i> = 2, compute upper bounds on <i>Difu</i> and <i>Difl</i>, using F-norm-based estimate (<i>dif</i> (1:2));</p> <p>If <i>ijob</i> = 3, compute estimate of <i>Difu</i> and <i>Difl</i>, using 1-norm-based estimate (<i>dif</i> (1:2)). This option is about 5 times as expensive as <i>ijob</i> = 2;</p> <p>If <i>ijob</i> = 4, compute <i>pl</i>, <i>pr</i> and <i>dif</i> (i.e., options 0, 1 and 2 above). This is an economic version to get it all;</p> <p>If <i>ijob</i> = 5, compute <i>pl</i>, <i>pr</i> and <i>dif</i> (i.e., options 0, 1 and 3 above).</p>
<i>wantq, wantz</i>	<p>If <i>wantq</i> = 1, update the left transformation matrix <i>Q</i>;</p> <p>If <i>wantq</i> = 0, do not update <i>Q</i>;</p> <p>If <i>wantz</i> = 1, update the right transformation matrix <i>Z</i>;</p> <p>If <i>wantz</i> = 0, do not update <i>Z</i>.</p>
<i>select</i>	<p>Array, size at least $\max(1, n)$. Specifies the eigenvalues in the selected cluster.</p> <p>To select an eigenvalue ω_j, <i>select</i>[<i>j</i> - 1] must be 1. For real flavors: to select a complex conjugate pair of eigenvalues ω_j and ω_{j+1} (corresponding 2 by 2 diagonal block), <i>select</i>[<i>j</i> - 1] and/or <i>select</i>[<i>j</i>] must be set to 1; the complex conjugate ω_j and ω_{j+1} must be either both included in the cluster or both excluded.</p>
<i>n</i>	The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>a, b, q, z</i>	<p>Arrays:</p> <p><i>a</i> (size $\max(1, lda * n)$) contains the matrix <i>A</i>.</p> <p>For real flavors: <i>A</i> is upper quasi-triangular, with (<i>A</i>, <i>B</i>) in generalized real Schur canonical form.</p> <p>For complex flavors: <i>A</i> is upper triangular, in generalized Schur canonical form.</p> <p><i>b</i> (size $\max(1, ldb * n)$) contains the matrix <i>B</i>.</p> <p>For real flavors: <i>B</i> is upper triangular, with (<i>A</i>, <i>B</i>) in generalized real Schur canonical form.</p>

For complex flavors: B is upper triangular, in generalized Schur canonical form.

q (size at least 1 if $wantq = 0$ and at least $\max(1, ldq*n)$ if $wantq = 1$)

If $wantq = 1$, then q is an n -by- n matrix;

If $wantq = 0$, then q is not referenced.

z (size at least 1 if $wantz = 0$ and at least $\max(1, ldz*n)$ if $wantz = 1$)

If $wantz = 1$, then z is an n -by- n matrix;

If $wantz = 0$, then z is not referenced.

lda The leading dimension of a ; at least $\max(1, n)$.

ldb The leading dimension of b ; at least $\max(1, n)$.

ldq The leading dimension of q ; $ldq \geq 1$.

If $wantq = 1$, then $ldq \geq \max(1, n)$.

ldz The leading dimension of z ; $ldz \geq 1$.

If $wantz = 1$, then $ldz \geq \max(1, n)$.

Output Parameters

a, b Overwritten by the reordered matrices A and B , respectively.

$alphar, alphai$ Arrays, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

$alpha$ Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors.

See *beta*.

$beta$ Array, size at least $\max(1, n)$.

For real flavors:

On exit, $(alphar[j] + alphai[j]*i)/beta[j]$, $j=0, \dots, n-1$, will be the generalized eigenvalues.

$alphar[j] + alphai[j]*i$ and $beta[j]$, $j=0, \dots, n-1$ are the diagonals of the complex Schur form (S,T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A,B) were further reduced to triangular form using complex unitary transformations.

If $alphai[j-1]$ is zero, then the j -th eigenvalue is real; if positive, then the j -th and $(j+1)$ -st eigenvalues are a complex conjugate pair, with $alphai[j]$ negative.

For complex flavors:

The diagonal elements of A and B , respectively, when the pair (A,B) has been reduced to generalized Schur form. $alpha[i]/beta[i]$, $i=0, \dots, n-1$ are the generalized eigenvalues.

<i>q</i>	If <i>wantq</i> = 1, then, on exit, <i>Q</i> has been postmultiplied by the left orthogonal transformation matrix which reorder (<i>A</i> , <i>B</i>). The leading <i>m</i> columns of <i>Q</i> form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).
<i>z</i>	If <i>wantz</i> = 1, then, on exit, <i>Z</i> has been postmultiplied by the left orthogonal transformation matrix which reorder (<i>A</i> , <i>B</i>). The leading <i>m</i> columns of <i>Z</i> form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).
<i>m</i>	The dimension of the specified pair of left and right eigen-spaces (deflating subspaces); $0 \leq m \leq n$.
<i>pl</i> , <i>pr</i>	<p>If <i>ijob</i> = 1, 4, or 5, <i>pl</i> and <i>pr</i> are lower bounds on the reciprocal of the norm of "projections" onto left and right eigenspaces with respect to the selected cluster.</p> <p>$0 < pl, pr \leq 1$. If <i>m</i> = 0 or <i>m</i> = <i>n</i>, <i>pl</i> = <i>pr</i> = 1.</p> <p>If <i>ijob</i> = 0, 2 or 3, <i>pl</i> and <i>pr</i> are not referenced</p>
<i>dif</i>	<p>Array, size (2).</p> <p>If <i>ijob</i> ≥ 2, <i>dif</i>(1:2) store the estimates of <i>Difu</i> and <i>Difl</i>.</p> <p>If <i>ijob</i> = 2 or 4, <i>dif</i>(1:2) are F-norm-based upper bounds on <i>Difu</i> and <i>Difl</i>.</p> <p>If <i>ijob</i> = 3 or 5, <i>dif</i>(1:2) are 1-norm-based estimates of <i>Difu</i> and <i>Difl</i>.</p> <p>If <i>m</i> = 0 or <i>m</i> = <i>n</i>, <i>dif</i>(1:2) = F-norm([<i>A</i>, <i>B</i>]).</p> <p>If <i>ijob</i> = 0 or 1, <i>dif</i> is not referenced.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, Reordering of (*A*, *B*) failed because the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is very ill-conditioned. (*A*, *B*) may have been partially reordered.

If *ijob* > 0, 0 is returned in *dif*, *pl* and *pr*.

?tgsyl

Solves the generalized Sylvester equation.

Syntax

```
lapack_int LAPACKE_stgsyl( int matrix_layout, char trans, lapack_int ijob, lapack_int
m, lapack_int n, const float* a, lapack_int lda, const float* b, lapack_int ldb, float*
c, lapack_int ldc, const float* d, lapack_int ldd, const float* e, lapack_int lde,
float* f, lapack_int ldf, float* scale, float* dif );
```

```
lapack_int LAPACKE_dtgsyl( int matrix_layout, char trans, lapack_int ijob, lapack_int
m, lapack_int n, const double* a, lapack_int lda, const double* b, lapack_int ldb,
double* c, lapack_int ldc, const double* d, lapack_int ldd, const double* e, lapack_int
lde, double* f, lapack_int ldf, double* scale, double* dif );
```

```

lapack_int LAPACKE_ctgsyl( int matrix_layout, char trans, lapack_int ijob, lapack_int
m, lapack_int n, const lapack_complex_float* a, lapack_int lda, const
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* c, lapack_int ldc, const
lapack_complex_float* d, lapack_int ldd, const lapack_complex_float* e, lapack_int lde,
lapack_complex_float* f, lapack_int ldf, float* scale, float* dif );

lapack_int LAPACKE_ztgsyl( int matrix_layout, char trans, lapack_int ijob, lapack_int
m, lapack_int n, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* c, lapack_int ldc,
const lapack_complex_double* d, lapack_int ldd, const lapack_complex_double* e,
lapack_int lde, lapack_complex_double* f, lapack_int ldf, double* scale, double* dif );

```

Include Files

- mkl.h

Description

The routine solves the generalized Sylvester equation:

$$A^*R - L^*B = scale^*C$$

$$D^*R - L^*E = scale^*F$$

where R and L are unknown m -by- n matrices, (A, D) , (B, E) and (C, F) are given matrix pairs of size m -by- m , n -by- n and m -by- n , respectively, with real/complex entries. (A, D) and (B, E) must be in generalized real-Schur/Schur canonical form, that is, A, B are upper quasi-triangular/triangular and D, E are upper triangular.

The solution (R, L) overwrites (C, F) . The factor $scale$, $0 \leq scale \leq 1$, is an output scaling factor chosen to avoid overflow.

In matrix notation the above equation is equivalent to the following: solve $Z^*x = scale^*b$, where Z is defined as

$$Z = \begin{pmatrix} kron(I_n, A) & -kron(B^T, I_m) \\ kron(I_n, D) & -kron(E^T, I_m) \end{pmatrix}$$

Here I_k is the identity matrix of size k and X^T is the transpose/conjugate-transpose of X . $kron(X, Y)$ is the Kronecker product between the matrices X and Y .

If $trans = 'T'$ (for real flavors), or $trans = 'C'$ (for complex flavors), the routine ?tgsyl solves the transposed/conjugate-transposed system $Z^T*y = scale*b$, which is equivalent to solve for R and L in

$$A^T*R + D^T*L = scale^*C$$

$$R^*B^T + L^*E^T = scale^*(-F)$$

This case ($trans = 'T'$ for stgsyl/dtgsyl or $trans = 'C'$ for ctgsyl/ztgsyl) is used to compute an one-norm-based estimate of $Dif[(A, D), (B, E)]$, the separation between the matrix pairs (A, D) and (B, E) .

If $ijob \geq 1$, ?tgsyl computes a Frobenius norm-based estimate of $Dif[(A, D), (B, E)]$. That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of Z . This is a level 3 BLAS algorithm.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>trans</i>	<p>Must be 'N', 'T', or 'C'.</p> <p>If <i>trans</i> = 'N', solve the generalized Sylvester equation.</p> <p>If <i>trans</i> = 'T', solve the 'transposed' system (for real flavors only).</p> <p>If <i>trans</i> = 'C', solve the 'conjugate transposed' system (for complex flavors only).</p>
<i>ijob</i>	<p>Specifies what kind of functionality to be performed:</p> <p>If <i>ijob</i> = 0, solve the generalized Sylvester equation only;</p> <p>If <i>ijob</i> = 1, perform the functionality of <i>ijob</i> = 0 and <i>ijob</i> = 3;</p> <p>If <i>ijob</i> = 2, perform the functionality of <i>ijob</i> = 0 and <i>ijob</i> = 4;</p> <p>If <i>ijob</i> = 3, only an estimate of $\text{Diff}[(A, D), (B, E)]$ is computed (look ahead strategy is used);</p> <p>If <i>ijob</i> = 4, only an estimate of $\text{Diff}[(A, D), (B, E)]$ is computed (?gecon on sub-systems is used). If <i>trans</i> = 'T' or 'C', <i>ijob</i> is not referenced.</p>
<i>m</i>	The order of the matrices <i>A</i> and <i>D</i> , and the row dimension of the matrices <i>C</i> , <i>F</i> , <i>R</i> and <i>L</i> .
<i>n</i>	The order of the matrices <i>B</i> and <i>E</i> , and the column dimension of the matrices <i>C</i> , <i>F</i> , <i>R</i> and <i>L</i> .
<i>a</i> , <i>b</i> , <i>c</i> , <i>d</i> , <i>e</i> , <i>f</i>	<p>Arrays:</p> <p><i>a</i> (size $\max(1, lda*m)$) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix <i>A</i>.</p> <p><i>b</i> (size $\max(1, ldb*n)$) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix <i>B</i>.</p> <p><i>c</i> (size $\max(1, ldc*n)$ for column major layout and $\max(1, ldc*m)$ for row major layout) contains the right-hand-side of the first matrix equation in the generalized Sylvester equation (as defined by <i>trans</i>)</p> <p><i>d</i> (size $\max(1, ldd*m)$) contains the upper triangular matrix <i>D</i>.</p> <p><i>e</i> (size $\max(1, lde*n)$) contains the upper triangular matrix <i>E</i>.</p> <p><i>f</i> (size $\max(1, ldf*n)$ for column major layout and $\max(1, ldf*m)$ for row major layout) contains the right-hand-side of the second matrix equation in the generalized Sylvester equation (as defined by <i>trans</i>)</p>
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; at least $\max(1, n)$.
<i>ldc</i>	The leading dimension of <i>c</i> ; at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout .
<i>ldd</i>	The leading dimension of <i>d</i> ; at least $\max(1, m)$.
<i>lde</i>	The leading dimension of <i>e</i> ; at least $\max(1, n)$.

ldf The leading dimension of *f*; at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout .

Output Parameters

c If *ijob*=0, 1, or 2, overwritten by the solution *R*.
If *ijob*=3 or 4 and *trans* = 'N', *c* holds *R*, the solution achieved during the computation of the Dif-estimate.

f If *ijob*=0, 1, or 2, overwritten by the solution *L*.
If *ijob*=3 or 4 and *trans* = 'N', *f* holds *L*, the solution achieved during the computation of the Dif-estimate.

dif On exit, *dif* is the reciprocal of a lower bound of the reciprocal of the Dif-function, that is, *dif* is an upper bound of $\text{Dif}[(A, D), (B, E)] = \sigma_{\min}(Z)$, where *Z* as defined in the description.
If *ijob* = 0, or *trans* = 'T' (for real flavors), or *trans* = 'C' (for complex flavors), *dif* is not touched.

scale On exit, *scale* is the scaling factor in the generalized Sylvester equation.
If $0 < \text{scale} < 1$, *c* and *f* hold the solutions *R* and *L*, respectively, to a slightly perturbed system but the input matrices *A*, *B*, *D* and *E* have not been changed.
If *scale* = 0, *c* and *f* hold the solutions *R* and *L*, respectively, to the homogeneous system with $C = F = 0$. Normally, *scale* = 1.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, (*A*, *D*) and (*B*, *E*) have common or close eigenvalues.

?tgsna

Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

Syntax

```
lapack_int LAPACKE_stgsna( int matrix_layout, char job, char howmny, const
lapack_logical* select, lapack_int n, const float* a, lapack_int lda, const float* b,
lapack_int ldb, const float* vl, lapack_int ldvl, const float* vr, lapack_int ldvr,
float* s, float* dif, lapack_int mm, lapack_int* m );
```

```
lapack_int LAPACKE_dtgsna( int matrix_layout, char job, char howmny, const
lapack_logical* select, lapack_int n, const double* a, lapack_int lda, const double* b,
lapack_int ldb, const double* vl, lapack_int ldvl, const double* vr, lapack_int ldvr,
double* s, double* dif, lapack_int mm, lapack_int* m );
```

```

lapack_int LAPACKE_ctgsna( int matrix_layout, char job, char howmny, const
lapack_logical* select, lapack_int n, const lapack_complex_float* a, lapack_int lda,
const lapack_complex_float* b, lapack_int ldb, const lapack_complex_float* vl,
lapack_int ldvl, const lapack_complex_float* vr, lapack_int ldvr, float* s, float* dif,
lapack_int mm, lapack_int* m );

lapack_int LAPACKE_ztgsna( int matrix_layout, char job, char howmny, const
lapack_logical* select, lapack_int n, const lapack_complex_double* a, lapack_int lda,
const lapack_complex_double* b, lapack_int ldb, const lapack_complex_double* vl,
lapack_int ldvl, const lapack_complex_double* vr, lapack_int ldvr, double* s, double*
dif, lapack_int mm, lapack_int* m );

```

Include Files

- mkl.h

Description

The real flavors `stgsna/dtgsna` of this routine estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair $(Q^*A^*Z^T, Q^*B^*Z^T)$ with orthogonal matrices Q and Z).

(A, B) must be in generalized real Schur form (as returned by [gges/gges](#)), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex flavors `ctgsna/ztgsna` estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) . (A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>job</code>	Specifies whether condition numbers are required for eigenvalues or eigenvectors. Must be 'E' or 'V' or 'B'. If <code>job</code> = 'E', for eigenvalues only (compute <code>s</code>). If <code>job</code> = 'V', for eigenvectors only (compute <code>dif</code>). If <code>job</code> = 'B', for both eigenvalues and eigenvectors (compute both <code>s</code> and <code>dif</code>).
<code>howmny</code>	Must be 'A' or 'S'. If <code>howmny</code> = 'A', compute condition numbers for all eigenpairs. If <code>howmny</code> = 'S', compute condition numbers for selected eigenpairs specified by the logical array <code>select</code> .
<code>select</code>	Array, size at least $\max(1, n)$. If <code>howmny</code> = 'S', <code>select</code> specifies the eigenpairs for which condition numbers are required. If <code>howmny</code> = 'A', <code>select</code> is not referenced. <i>For real flavors:</i>

To select condition numbers for the eigenpair corresponding to a real eigenvalue ω_j , *select*[*j* - 1] must be set to 1; to select condition numbers corresponding to a complex conjugate pair of eigenvalues ω_j and ω_{j+1} , either *select*[*j* - 1] or *select*[*j*] must be set to 1.

For complex flavors:

To select condition numbers for the corresponding *j*-th eigenvalue and/or eigenvector, *select*[*j* - 1] must be set to 1.

n

The order of the square matrix pair (*A*, *B*) ($n \geq 0$).

a, *b*, *vl*, *vr*

Arrays:

a (size $\max(1, lda * n)$) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix *A* in the pair (*A*, *B*).

b (size $\max(1, ldb * n)$) contains the upper triangular matrix *B* in the pair (*A*, *B*).

If *job* = 'E' or 'B', *vl* (size $\max(1, ldvl * m)$ for column major layout and $\max(1, ldvl * n)$ for row major layout) must contain left eigenvectors of (*A*, *B*), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by ?tgevc.

If *job* = 'V', *vl* is not referenced.

If *job* = 'E' or 'B', *vr* (size $\max(1, ldvr * m)$ for column major layout and $\max(1, ldvr * n)$ for row major layout) must contain right eigenvectors of (*A*, *B*), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by ?tgevc.

If *job* = 'V', *vr* is not referenced.

lda

The leading dimension of *a*; at least $\max(1, n)$.

ldb

The leading dimension of *b*; at least $\max(1, n)$.

ldvl

The leading dimension of *vl*; $ldvl \geq 1$.

If *job* = 'E' or 'B', then $ldvl \geq \max(1, n)$ for column major layout and $ldvl \geq \max(1, m)$ for row major layout.

ldvr

The leading dimension of *vr*; $ldvr \geq 1$.

If *job* = 'E' or 'B', then $ldvr \geq \max(1, n)$ for column major layout and $ldvr \geq \max(1, m)$ for row major layout.

mm

The number of elements in the arrays *s* and *dif* ($mm \geq m$).

Output Parameters

s

Array, size *mm*.

If *job* = 'E' or 'B', contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array.

If *job* = 'V', *s* is not referenced.

For real flavors:

For a complex conjugate pair of eigenvalues two consecutive elements of s are set to the same value. Thus, $s[j - 1]$, $dif[j - 1]$, and the j -th columns of vl and vr all correspond to the same eigenpair (but not in general the j -th eigenpair, unless all eigenpairs are selected).

dif

Array, size mm .

If $job = 'V'$ or $'B'$, contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array.

If the eigenvalues cannot be reordered to compute $dif[j]$, $dif[j]$ is set to 0; this can only occur when the true value would be very small anyway.

If $job = 'E'$, dif is not referenced.

For real flavors:

For a complex eigenvector, two consecutive elements of dif are set to the same value.

For complex flavors:

For each eigenvalue/vector specified by $select$, dif stores a Frobenius norm-based estimate of $Difl$.

m

The number of elements in the arrays s and dif used to store the specified condition numbers; for each selected eigenvalue one element is used.

If $howmny = 'A'$, m is set to n .

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Generalized Singular Value Decomposition: LAPACK Computational Routines

This topic describes LAPACK computational routines used for finding the generalized singular value decomposition (GSVD) of two matrices A and B as

$$U^H A Q = D_1 * (0 \ R),$$

$$V^H B Q = D_2 * (0 \ R),$$

where U , V , and Q are orthogonal/unitary matrices, R is a nonsingular upper triangular matrix, and D_1 , D_2 are "diagonal" matrices of the structure detailed in the routines description section.

Table "Computational Routines for Generalized Singular Value Decomposition" lists LAPACK routines that perform generalized singular value decomposition of matrices.

Computational Routines for Generalized Singular Value Decomposition

Routine name	Operation performed
ggsvp	Computes the preprocessing decomposition for the generalized SVD
ggsvp3	Performs preprocessing for a generalized SVD.
ggsvd3	Computes generalized SVD.

Routine name	Operation performed
tgsja	Computes the generalized SVD of two upper triangular or trapezoidal matrices

You can use routines listed in the above table as well as the driver routine [ggsvd](#) to find the GSVD of a pair of general rectangular matrices.

[?ggsvp](#)

Computes the preprocessing decomposition for the generalized SVD (deprecated).

Syntax

```
lapack_int LAPACKE_sggsvp( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, float* a, lapack_int lda, float* b, lapack_int
ldb, float tola, float tolb, lapack_int* k, lapack_int* l, float* u, lapack_int ldu,
float* v, lapack_int ldv, float* q, lapack_int ldq );
```

```
lapack_int LAPACKE_dggsvp( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, double* a, lapack_int lda, double* b,
lapack_int ldb, double tola, double tolb, lapack_int* k, lapack_int* l, double* u,
lapack_int ldu, double* v, lapack_int ldv, double* q, lapack_int ldq );
```

```
lapack_int LAPACKE_cggsvp( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_complex_float* a, lapack_int lda,
lapack_complex_float* b, lapack_int ldb, float tola, float tolb, lapack_int* k,
lapack_int* l, lapack_complex_float* u, lapack_int ldu, lapack_complex_float* v,
lapack_int ldv, lapack_complex_float* q, lapack_int ldq );
```

```
lapack_int LAPACKE_zggsvp( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_complex_double* a, lapack_int lda,
lapack_complex_double* b, lapack_int ldb, double tola, double tolb, lapack_int* k,
lapack_int* l, lapack_complex_double* u, lapack_int ldu, lapack_complex_double* v,
lapack_int ldv, lapack_complex_double* q, lapack_int ldq );
```

Include Files

- `mk1.h`

Description

This routine is deprecated; use [ggsvp3](#).

The routine computes orthogonal matrices U , V and Q such that

$$U^H A Q = \begin{matrix} & n-k-l & k & l \\ & k & & \\ & l & & \\ m-k-l & & & \end{matrix} \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{matrix} & n-k-l & k & l \\ & k & \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix} & \end{matrix}, \quad \text{if } m-k-l < 0$$

$$V^H B Q = \begin{matrix} & n-k-l & k & l \\ p-l & \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix} & \end{matrix}$$

where the k -by- k matrix A_{12} and l -by- l matrix B_{13} are nonsingular upper triangular; A_{23} is l -by- l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$ -by- l upper trapezoidal. The sum $k+l$ is equal to the effective numerical rank of the $(m+p)$ -by- n matrix $(A^H, B^H)^H$.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine [?tgsja](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobu</i>	Must be 'U' or 'N'. If <i>jobu</i> = 'U', orthogonal/unitary matrix U is computed. If <i>jobu</i> = 'N', U is not computed.
<i>jobv</i>	Must be 'V' or 'N'. If <i>jobv</i> = 'V', orthogonal/unitary matrix V is computed. If <i>jobv</i> = 'N', V is not computed.
<i>jobq</i>	Must be 'Q' or 'N'. If <i>jobq</i> = 'Q', orthogonal/unitary matrix Q is computed. If <i>jobq</i> = 'N', Q is not computed.
<i>m</i>	The number of rows of the matrix A ($m \geq 0$).
<i>p</i>	The number of rows of the matrix B ($p \geq 0$).
<i>n</i>	The number of columns of the matrices A and B ($n \geq 0$).
<i>a, b</i>	Arrays: a (size at least $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout) contains the m -by- n matrix A .

b (size at least $\max(1, ldb*n)$ for column major layout and $\max(1, ldb*p)$ for row major layout) contains the p -by- n matrix B .

lda

The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

ldb

The leading dimension of b ; at least $\max(1, p)$ for column major layout and $\max(1, n)$ for row major layout.

tol_a, tol_b

tol_a and tol_b are the thresholds to determine the effective numerical rank of matrix B and a subblock of A . Generally, they are set to

$$tol_a = \max(m, n) * ||A|| * \text{MACHEPS},$$

$$tol_b = \max(p, n) * ||B|| * \text{MACHEPS}.$$

The size of tol_a and tol_b may affect the size of backward errors of the decomposition.

ldu

The leading dimension of the output array u . $ldu \geq \max(1, m)$ if $jobu = 'U'$; $ldu \geq 1$ otherwise.

ldv

The leading dimension of the output array v . $ldv \geq \max(1, p)$ if $jobv = 'V'$; $ldv \geq 1$ otherwise.

ldq

The leading dimension of the output array q . $ldq \geq \max(1, n)$ if $jobq = 'Q'$; $ldq \geq 1$ otherwise.

Output Parameters

a

Overwritten by the triangular (or trapezoidal) matrix described in the *Description* section.

b

Overwritten by the triangular matrix described in the *Description* section.

k, l

On exit, k and l specify the dimension of subblocks. The sum $k + l$ is equal to effective numerical rank of $(A^H, B^H)^H$.

u, v, q

Arrays:

If $jobu = 'U'$, u (size $\max(1, ldu*m)$) contains the orthogonal/unitary matrix U .

If $jobu = 'N'$, u is not referenced.

If $jobv = 'V'$, v (size $\max(1, ldv*p)$) contains the orthogonal/unitary matrix V .

If $jobv = 'N'$, v is not referenced.

If $jobq = 'Q'$, q (size $\max(1, ldq*n)$) contains the orthogonal/unitary matrix Q .

If $jobq = 'N'$, q is not referenced.

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

?ggsvp3*Performs preprocessing for a generalized SVD.***Syntax**

```

lapack_int LAPACKE_sggsvp3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, float * a, lapack_int lda, float * b,
lapack_int ldb, float tola, float tolb, lapack_int * k, lapack_int * l, float * u,
lapack_int ldu, float * v, lapack_int ldv, float * q, lapack_int ldq);

lapack_int LAPACKE_dggsvp3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, double * a, lapack_int lda, double * b,
lapack_int ldb, double tola, double tolb, lapack_int * k, lapack_int * l, double * u,
lapack_int ldu, double * v, lapack_int ldv, double * q, lapack_int ldq);

lapack_int LAPACKE_cggsvp3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_complex_float * a, lapack_int lda,
lapack_complex_float * b, lapack_int ldb, float tola, float tolb, lapack_int * k,
lapack_int * l, lapack_complex_float * u, lapack_int ldu, lapack_complex_float * v,
lapack_int ldv, lapack_complex_float * q, lapack_int ldq);

lapack_int LAPACKE_zggsvp3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_complex_double * a, lapack_int lda,
lapack_complex_double * b, lapack_int ldb, double tola, double tolb, lapack_int * k,
lapack_int * l, lapack_complex_double * u, lapack_int ldu, lapack_complex_double * v,
lapack_int ldv, lapack_complex_double * q, lapack_int ldq);

```

Include Files

- mkl_lapack.h

Include Files

- mkl.h

Description

?ggsvp3 computes orthogonal or unitary matrices U , V , and Q such that

for real flavors:

$$U^T A Q = \begin{matrix} & & n-k-l & k & l \\ & k & & & \\ & l & & & \\ m-k-l & & \begin{pmatrix} 0 & A12 & A13 \\ 0 & 0 & A23 \\ 0 & 0 & 0 \end{pmatrix} \end{matrix} \text{ if } m-k-l \geq 0;$$

$$U^T A Q = \begin{matrix} & & n-k-l & k & l \\ & k & & & \\ m-k & & \begin{pmatrix} 0 & A12 & A13 \\ 0 & 0 & A23 \end{pmatrix} \end{matrix} \text{ if } m-k-l < 0;$$

$$V^T B Q = \begin{matrix} & & n-k-l & k & l \\ & l & & & \\ p-l & & \begin{pmatrix} 0 & 0 & B13 \\ 0 & 0 & 0 \end{pmatrix} \end{matrix}$$

for complex flavors:

$$U^H A Q = \begin{matrix} & & n-k-l & k & l \\ & k & & & \\ & l & & & \\ m-k-l & & \begin{pmatrix} 0 & A12 & A13 \\ 0 & 0 & A23 \\ 0 & 0 & 0 \end{pmatrix} \end{matrix} \text{ if } m-k-l \geq 0;$$

$$U^H A Q = \begin{matrix} & n-k-l & k & l \\ & k & \begin{pmatrix} 0 & A12 & A13 \\ 0 & 0 & A23 \end{pmatrix} \end{matrix} \text{ if } m-k-l < 0;$$

$$V^H B Q = \begin{matrix} & n-k-l & k & l \\ l & \begin{pmatrix} 0 & 0 & B13 \\ 0 & 0 & 0 \end{pmatrix} \\ p-l & \end{matrix}$$

where the k -by- k matrix $A12$ and l -by- l matrix $B13$ are nonsingular upper triangular; $A23$ is l -by- l upper triangular if $m-k-l \geq 0$, otherwise $A23$ is $(m-k)$ -by- l upper trapezoidal. $k + l$ = the effective numerical rank of the $(m + p)$ -by- n matrix $(A^T, B^T)^T$ for real flavors or $(A^H, B^H)^H$ for complex flavors.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see ?ggsvd3.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>jobu</code>	= 'U': Orthogonal/unitary matrix U is computed; = 'N': U is not computed.
<code>jobv</code>	= 'V': Orthogonal/unitary matrix V is computed; = 'N': V is not computed.
<code>jobq</code>	= 'Q': Orthogonal/unitary matrix Q is computed; = 'N': Q is not computed.
<code>m</code>	The number of rows of the matrix A . $m \geq 0$.
<code>p</code>	The number of rows of the matrix B . $p \geq 0$.
<code>n</code>	The number of columns of the matrices A and B . $n \geq 0$.
<code>a</code>	Array, size $(lda * n)$. On entry, the m -by- n matrix A .
<code>lda</code>	The leading dimension of the array a . $lda \geq \max(1, m)$.
<code>b</code>	Array, size $(ldb * n)$. On entry, the p -by- n matrix B .
<code>ldb</code>	The leading dimension of the array b . $ldb \geq \max(1, p)$.
<code>tola, tol b</code>	$tola$ and $tol b$ are the thresholds to determine the effective numerical rank of matrix B and a subblock of A . Generally, they are set to $tola = \max(m, n) * \text{norm}(a) * \text{MACHEPS}$,

$tolb = \max(p,n)*\text{norm}(b)*\text{MACHEPS}$.

The size of tol_a and tol_b may affect the size of backward errors of the decomposition.

ldu

The leading dimension of the array u .

$ldu \geq \max(1,m)$ if $jobu = 'U'$; $ldu \geq 1$ otherwise.

ldv

The leading dimension of the array v .

$ldv \geq \max(1,p)$ if $jobv = 'V'$; $ldv \geq 1$ otherwise.

ldq

The leading dimension of the array q .

$ldq \geq \max(1,n)$ if $jobq = 'Q'$; $ldq \geq 1$ otherwise.

Output Parameters

a

On exit, a contains the triangular (or trapezoidal) matrix described in the Description section.

b

On exit, b contains the triangular matrix described in the Description section.

k, l

On exit, k and l specify the dimension of the subblocks described in Description section.

$k + l = \text{effective numerical rank of } (A^T, B^T)^T \text{ for real flavors or } (A^H, B^H)^H \text{ for complex flavors.}$

u

Array, size $(ldu*m)$.

If $jobu = 'U'$, u contains the orthogonal/unitary matrix U .

If $jobu = 'N'$, u is not referenced.

v

Array, size $(ldv*p)$.

If $jobv = 'V'$, v contains the orthogonal/unitary matrix V .

If $jobv = 'N'$, v is not referenced.

q

Array, size $(ldq*n)$.

If $jobq = 'Q'$, q contains the orthogonal/unitary matrix Q .

If $jobq = 'N'$, q is not referenced.

Return Values

This function returns a value $info$.

= 0: successful exit.

< 0: if $info = -i$, the i -th argument had an illegal value.

Application Notes

The subroutine uses LAPACK subroutine `?geqp3` for the QR factorization with column pivoting to detect the effective numerical rank of the A matrix. It may be replaced by a better rank determination strategy.

`?ggsvp3` replaces the deprecated subroutine `?ggsvp`.

?ggsvd3*Computes generalized SVD.***Syntax**

```
lapack_int LAPACKE_sggsvd3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int * k, lapack_int * l, float * a,
lapack_int lda, float * b, lapack_int ldb, float * alpha, float * beta, float * u,
lapack_int ldu, float * v, lapack_int ldv, float * q, lapack_int ldq, lapack_int *
iwork);
```

```
lapack_int LAPACKE_dggsvd3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int * k, lapack_int * l, double * a,
lapack_int lda, double * b, lapack_int ldb, double * alpha, double * beta, double * u,
lapack_int ldu, double * v, lapack_int ldv, double * q, lapack_int ldq, lapack_int *
iwork);
```

```
lapack_int LAPACKE_cggsvd3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int * k, lapack_int * l,
lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb,
float * alpha, float * beta, lapack_complex_float * u, lapack_int ldu,
lapack_complex_float * v, lapack_int ldv, lapack_complex_float * q, lapack_int ldq,
lapack_int * iwork);
```

```
lapack_int LAPACKE_zggsvd3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int * k, lapack_int * l,
lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb,
double * alpha, double * beta, lapack_complex_double * u, lapack_int ldu,
lapack_complex_double * v, lapack_int ldv, lapack_complex_double * q, lapack_int ldq,
lapack_int * iwork);
```

Include Files

- mkl.h

Description

?ggsvd3 computes the generalized singular value decomposition (GSVD) of an m -by- n real or complex matrix A and p -by- n real or complex matrix B :

$$U^T A Q = D_1 \begin{pmatrix} 0 & R \end{pmatrix}, \quad V^T B Q = D_2 \begin{pmatrix} 0 & R \end{pmatrix} \text{ for real flavors}$$

or

$$U^H A Q = D_1 \begin{pmatrix} 0 & R \end{pmatrix}, \quad V^H B Q = D_2 \begin{pmatrix} 0 & R \end{pmatrix} \text{ for complex flavors}$$

where U , V and Q are orthogonal/unitary matrices.

Let $k+1$ = the effective numerical rank of the matrix $(A^T B^T)^T$ for real flavors or the matrix $(A^H B^H)^H$ for complex flavors, then R is a $(k+1)$ -by- $(k+1)$ nonsingular upper triangular matrix, D_1 and D_2 are m -by- $(k+1)$ and p -by- $(k+1)$ "diagonal" matrices and of the following structures, respectively:

If $m-k-1 \geq 0$,

$$D_1 = \begin{matrix} & & k & l \\ & k & & \\ & l & & \\ m-k-l & & \begin{pmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{matrix}$$

$$D_2 = \begin{matrix} & k & l \\ l & \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \\ p-l & \end{matrix}$$

$$\begin{pmatrix} 0 & R \end{pmatrix} = \begin{matrix} & n-k-l & k & l \\ k & \begin{pmatrix} 0 & R11 & R12 \\ l & 0 & R22 \end{pmatrix} \end{matrix}$$

where

$$C = \text{diag}(\alpha(k+1), \dots, \alpha(k+l)),$$

$$S = \text{diag}(\beta(k+1), \dots, \beta(k+l)),$$

$$C^2 + S^2 = I.$$

If $m - k - l < 0$,

$$D_1 = \begin{matrix} & k & m-k & k+l-m \\ m-k & \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{matrix}$$

$$D_2 = \begin{matrix} & k & m-k & k+l-m \\ m-k & \begin{pmatrix} 0 & S & 0 \\ k+l-m & 0 & I \\ p-l & 0 & 0 \end{pmatrix} \end{matrix}$$

$$\begin{pmatrix} 0 & R \end{pmatrix} = \begin{matrix} & n-k-l & k & m-k & k+l-m \\ k & \begin{pmatrix} 0 & R11 & R12 & R13 \\ m-k & 0 & R22 & R23 \\ k+l-m & 0 & 0 & R33 \end{pmatrix} \end{matrix}$$

where

$$C = \text{diag}(\alpha(k+1), \dots, \alpha(m)),$$

$$S = \text{diag}(\beta(k+1), \dots, \beta(m)),$$

$$C^2 + S^2 = I.$$

The routine computes C , S , R , and optionally the orthogonal/unitary transformation matrices U , V and Q .

In particular, if B is an n -by- n nonsingular matrix, then the GSVD of A and B implicitly gives the SVD of $A \cdot \text{inv}(B)$:

$$A \cdot \text{inv}(B) = U \cdot (D_1 \cdot \text{inv}(D_2)) \cdot V^T \text{ for real flavors}$$

or

$$A \cdot \text{inv}(B) = U \cdot (D_1 \cdot \text{inv}(D_2)) \cdot V^H \text{ for complex flavors.}$$

If $(A^T, B^T)^T$ for real flavors or $(A^H, B^H)^H$ for complex flavors has orthonormal columns, then the GSVD of A and B is also equal to the CS decomposition of A and B . Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

$$A^T \cdot A \cdot X = \lambda \cdot B^T \cdot B \cdot X \text{ for real flavors}$$

or

$$A^H \cdot A \cdot X = \lambda \cdot B^H \cdot B \cdot X \text{ for complex flavors}$$

In some literature, the GSVD of A and B is presented in the form

$$U^T \cdot A \cdot X = (0 \ D_1), \quad V^T \cdot B \cdot X = (0 \ D_2) \text{ for real } (A, B)$$

or

$U^H A X = (0 \ D_1)$, $V^H B X = (0 \ D_2)$ for complex (A, B)

where U and V are orthogonal and X is nonsingular, D_1 and D_2 are "diagonal". The former GSVD form can be converted to the latter form by taking the nonsingular matrix X as

$$X = Q^* \begin{pmatrix} I & 0 \\ 0 & inv(R) \end{pmatrix}$$

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobu</i>	= 'U': Orthogonal/unitary matrix U is computed; = 'N': U is not computed.
<i>jobv</i>	= 'V': Orthogonal/unitary matrix V is computed; = 'N': V is not computed.
<i>jobq</i>	= 'Q': Orthogonal/unitary matrix Q is computed; = 'N': Q is not computed.
<i>m</i>	The number of rows of the matrix A . $m \geq 0$.
<i>n</i>	The number of columns of the matrices A and B . $n \geq 0$.
<i>p</i>	The number of rows of the matrix B . $p \geq 0$.
<i>a</i>	Array, size ($lda * n$). On entry, the m -by- n matrix A .
<i>lda</i>	The leading dimension of the array a . $lda \geq \max(1, m)$.
<i>b</i>	Array, size ($ldb * n$). On entry, the p -by- n matrix B .
<i>ldb</i>	The leading dimension of the array b . $ldb \geq \max(1, p)$.
<i>ldu</i>	The leading dimension of the array u . $ldu \geq \max(1, m)$ if <i>jobu</i> = 'U'; $ldu \geq 1$ otherwise.
<i>ldv</i>	The leading dimension of the array v . $ldv \geq \max(1, p)$ if <i>jobv</i> = 'V'; $ldv \geq 1$ otherwise.
<i>ldq</i>	The leading dimension of the array q . $ldq \geq \max(1, n)$ if <i>jobq</i> = 'Q'; $ldq \geq 1$ otherwise.
<i>iwork</i>	Array, size (n).

Output Parameters

k, l	<p>On exit, k and l specify the dimension of the subblocks described in the Description section.</p> <p>$k + l$ = effective numerical rank of $(A^T, B^T)^T$ for real flavors or $(A^H, B^H)^H$ for complex flavors.</p>
a	<p>On exit, a contains the triangular matrix R, or part of R.</p> <p>If $m - k - l \geq 0$, R is stored in the elements of array a corresponding to A_1: $k + l, n - k - l + 1 : n$.</p> <p>If $m - k - l < 0$, $\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$ is stored in the elements of array a corresponding to $A_{(1:m, n - k - l + 1 : n)}$, and R_{33} is stored in the elements of array a corresponding to $A_{m - k + 1 : l, n + m - k - l + 1 : n}$ on exit.</p>
b	<p>On exit, b contains part of the triangular matrix R if $m - k - l < 0$. See Description for details.</p>
$alpha$	Array, size (n)
$beta$	<p>Array, size (n)</p> <p>On exit, $alpha$ and $beta$ contain the generalized singular value pairs of a and b;</p> <p>$alpha[0 : k - 1] = 1$, $beta[0 : k - 1] = 0$, and if $m - k - l \geq 0$, $alpha[k : k + l - 1] = C$, $beta[k : k + l - 1] = S$, or if $m - k - l < 0$, $alpha[k : m - 1] = C$, $alpha[m : k + l - 1] = 0$ $beta[k : m - 1] = S$, $beta[m : k + l - 1] = 1$ and $alpha[k + l : n - 1] = 0$ $beta[k + l : n - 1] = 0$</p>
u	<p>Array, size ($ldu * m$).</p> <p>If $jobu = 'U'$, u contains the m-by-m orthogonal/unitary matrix U. If $jobu = 'N'$, u is not referenced.</p>
v	<p>Array, size ($ldv * p$).</p> <p>If $jobv = 'V'$, v contains the p-by-p orthogonal/unitary matrix V. If $jobv = 'N'$, v is not referenced.</p>
q	Array, size ($ldq * n$).

If *jobq* = 'Q', *q* contains the *n*-by-*n* orthogonal/unitary matrix *Q*.

If *jobq* = 'N', *q* is not referenced.

iwork

On exit, *iwork* stores the sorting information. More precisely, the following loop uses *iwork* to sort *alpha*:

```
for (i = k; i < min(m, k + 1); i++) {
    swap (alpha[i], alpha[iwork[i] - 1]);
}
```

such that $\alpha[0] \geq \alpha[1] \geq \dots \geq \alpha[n - 1]$.

Return Values

This function returns a value *info*.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

> 0: if *info* = 1, the Jacobi-type procedure failed to converge.

For further details, see subroutine ?tgsja.

Application Notes

?ggsvd3 replaces the deprecated subroutine ?ggsvd.

?tgsja

Computes the generalized SVD of two upper triangular or trapezoidal matrices.

Syntax

```
lapack_int LAPACKE_stgsja( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_int k, lapack_int l, float* a,
lapack_int lda, float* b, lapack_int ldb, float tola, float tolb, float* alpha, float*
beta, float* u, lapack_int ldu, float* v, lapack_int ldv, float* q, lapack_int ldq,
lapack_int* ncycle );
```

```
lapack_int LAPACKE_dtgsja( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_int k, lapack_int l, double* a,
lapack_int lda, double* b, lapack_int ldb, double tola, double tolb, double* alpha,
double* beta, double* u, lapack_int ldu, double* v, lapack_int ldv, double* q,
lapack_int ldq, lapack_int* ncycle );
```

```
lapack_int LAPACKE_ctgsja( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_int k, lapack_int l,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, float
tola, float tolb, float* alpha, float* beta, lapack_complex_float* u, lapack_int ldu,
lapack_complex_float* v, lapack_int ldv, lapack_complex_float* q, lapack_int ldq,
lapack_int* ncycle );
```

```
lapack_int LAPACKE_ztgsja( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_int k, lapack_int l,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
double tola, double tolb, double* alpha, double* beta, lapack_complex_double* u,
lapack_int ldu, lapack_complex_double* v, lapack_int ldv, lapack_complex_double* q,
lapack_int ldq, lapack_int* ncycle );
```


Include Files

- mkl.h

Description

The routine computes the generalized singular value decomposition (GSVD) of two real/complex upper triangular (or trapezoidal) matrices A and B . On entry, it is assumed that matrices A and B have the following forms, which may be obtained by the preprocessing subroutine [ggsvp](#) from a general m -by- n matrix A and p -by- n matrix B :

$$A = \begin{matrix} & n-k-l & k & l \\ & k & & \\ & l & & \\ m-k-l & & & \end{matrix} \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix}, \quad \text{if } m-k-l \geq 0$$

$$= \begin{matrix} & n-k-l & k & l \\ & k & & \\ m-k & & & \end{matrix} \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix}, \quad \text{if } m-k-l < 0$$

$$B = \begin{matrix} & n-k-l & k & l \\ & l & & \\ p-l & & & \end{matrix} \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix}$$

where the k -by- k matrix A_{12} and l -by- l matrix B_{13} are nonsingular upper triangular; A_{23} is l -by- l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$ -by- l upper trapezoidal.

On exit,

$$U^H * A * Q = D_1 * (0 \ R), \quad V^H * B * Q = D_2 * (0 \ R),$$

where U , V and Q are orthogonal/unitary matrices, R is a nonsingular upper triangular matrix, and D_1 and D_2 are "diagonal" matrices, which are of the following structures:

If $m-k-l \geq 0$,

$$D_1 = \begin{matrix} & k & l \\ \begin{matrix} k \\ l \\ m-k-l \end{matrix} & \begin{pmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{matrix}$$

$$D_2 = \begin{matrix} & k & l \\ \begin{matrix} 1 \\ p-1 \\ n-k-l \end{matrix} & \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{matrix}$$

$$(0 \ R) = \begin{matrix} & k & l \\ \begin{matrix} k \\ 1 \end{matrix} & \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \end{matrix}$$

where

$C = \text{diag}(\alpha[k], \dots, \alpha[k+l-1])$

$S = \text{diag}(\beta[k], \dots, \beta[k+l-1])$

$C^2 + S^2 = I$

R is stored in $a(1:k+l, n-k-l+1:n)$ on exit.

If $m-k-l < 0$,

$$\begin{aligned}
 & \begin{matrix} & k & m-k & k+l-m \\ \begin{matrix} k \\ m-k \end{matrix} & \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{matrix} \\
 D_2 = & \begin{matrix} & k & m-k & k+l-m \\ \begin{matrix} m-k \\ k+l-m \\ p-l \end{matrix} & \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{matrix} \\
 (0 \ R) = & \begin{matrix} & n-k-l & k & m-k & k+l-m \\ \begin{matrix} k \\ m-k \\ k+l-m \end{matrix} & \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \end{matrix}
 \end{aligned}$$

where

$C = \text{diag}(\alpha[k], \dots, \alpha[m-1]),$

$S = \text{diag}(\beta[k], \dots, \beta[m-1]),$

$C^2 + S^2 = I$

On exit,

$$\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$$

is stored in $a(1:m, n-k-l+1:n)$ and R_{33} is stored

in $b(m-k+1:l, n+m-k-l+1:n)$.

The computation of the orthogonal/unitary transformation matrices U , V or Q is optional. These matrices may either be formed explicitly, or they may be postmultiplied into input matrices U_1 , V_1 , or Q_1 .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobu</i>	<p>Must be 'U', 'I', or 'N'.</p> <p>If <i>jobu</i> = 'U', <i>u</i> must contain an orthogonal/unitary matrix U_1 on entry.</p> <p>If <i>jobu</i> = 'I', <i>u</i> is initialized to the unit matrix.</p> <p>If <i>jobu</i> = 'N', <i>u</i> is not computed.</p>
<i>jobv</i>	<p>Must be 'V', 'I', or 'N'.</p> <p>If <i>jobv</i> = 'V', <i>v</i> must contain an orthogonal/unitary matrix V_1 on entry.</p> <p>If <i>jobv</i> = 'I', <i>v</i> is initialized to the unit matrix.</p> <p>If <i>jobv</i> = 'N', <i>v</i> is not computed.</p>
<i>jobq</i>	<p>Must be 'Q', 'I', or 'N'.</p> <p>If <i>jobq</i> = 'Q', <i>q</i> must contain an orthogonal/unitary matrix Q_1 on entry.</p> <p>If <i>jobq</i> = 'I', <i>q</i> is initialized to the unit matrix.</p> <p>If <i>jobq</i> = 'N', <i>q</i> is not computed.</p>
<i>m</i>	The number of rows of the matrix <i>A</i> ($m \geq 0$).
<i>p</i>	The number of rows of the matrix <i>B</i> ($p \geq 0$).
<i>n</i>	The number of columns of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>k, l</i>	Specify the subblocks in the input matrices <i>A</i> and <i>B</i> , whose GSVD is computed.
<i>a, b, u, v, q</i>	<p>Arrays:</p> <p><i>a</i>(size at least $\max(1, lda \cdot n)$ for column major layout and $\max(1, lda \cdot m)$ for row major layout) contains the <i>m</i>-by-<i>n</i> matrix <i>A</i>.</p> <p><i>b</i>(size at least $\max(1, ldb \cdot n)$ for column major layout and $\max(1, ldb \cdot p)$ for row major layout) contains the <i>p</i>-by-<i>n</i> matrix <i>B</i>.</p> <p>If <i>jobu</i> = 'U', <i>u</i> (size $\max(1, ldu \cdot m)$) must contain a matrix U_1 (usually the orthogonal/unitary matrix returned by <code>?ggsvp</code>).</p> <p>If <i>jobv</i> = 'V', <i>v</i> (size at least $\max(1, ldv \cdot p)$) must contain a matrix V_1 (usually the orthogonal/unitary matrix returned by <code>?ggsvp</code>).</p> <p>If <i>jobq</i> = 'Q', <i>q</i> (size at least $\max(1, ldq \cdot n)$) must contain a matrix Q_1 (usually the orthogonal/unitary matrix returned by <code>?ggsvp</code>).</p>
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
<i>ldb</i>	The leading dimension of <i>b</i> ; at least $\max(1, p)$ for column major layout and $\max(1, n)$ for row major layout.
<i>ldu</i>	<p>The leading dimension of the array <i>u</i>.</p> <p>$ldu \geq \max(1, m)$ if <i>jobu</i> = 'U'; $ldu \geq 1$ otherwise.</p>

<i>ldv</i>	The leading dimension of the array <i>v</i> . <i>ldv</i> ≥ max(1, <i>p</i>) if <i>jobv</i> = 'V'; <i>ldv</i> ≥ 1 otherwise.
<i>ldq</i>	The leading dimension of the array <i>q</i> . <i>ldq</i> ≥ max(1, <i>n</i>) if <i>jobq</i> = 'Q'; <i>ldq</i> ≥ 1 otherwise.
<i>tola</i> , <i>tolb</i>	<i>tola</i> and <i>tolb</i> are the convergence criteria for the Jacobi-Kogbetliantz iteration procedure. Generally, they are the same as used in ?ggsvp: $tola = \max(m, n) * A * \text{MACHEPS},$ $tolb = \max(p, n) * B * \text{MACHEPS}.$

Output Parameters

<i>a</i>	On exit, <i>a</i> (<i>n</i> - <i>k</i> +1: <i>n</i> , 1:min(<i>k</i> +1, <i>m</i>)) contains the triangular matrix <i>R</i> or part of <i>R</i> .
<i>b</i>	On exit, if necessary, <i>b</i> (<i>m</i> - <i>k</i> +1: <i>l</i> , <i>n</i> + <i>m</i> - <i>k</i> - <i>l</i> +1: <i>n</i>)) contains a part of <i>R</i> .
<i>alpha</i> , <i>beta</i>	Arrays, size at least max(1, <i>n</i>). Contain the generalized singular value pairs of <i>A</i> and <i>B</i> : $alpha(1:k) = 1,$ $beta(1:k) = 0,$ and if $m-k-l \geq 0,$ $alpha(k+1:k+l) = \text{diag}(C),$ $beta(k+1:k+l) = \text{diag}(S),$ or if $m-k-l < 0,$ $alpha(k+1:m) = \text{diag}(C), alpha(m+1:k+l) = 0$ $beta(k+1:m) = \text{diag}(S),$ $beta(m+1:k+l) = 1.$ Furthermore, if $k+l < n,$ $alpha(k+l+1:n) = 0$ and $beta(k+l+1:n) = 0.$
<i>u</i>	If <i>jobu</i> = 'I', <i>u</i> contains the orthogonal/unitary matrix <i>U</i> . If <i>jobu</i> = 'U', <i>u</i> contains the product $U_1 * U$. If <i>jobu</i> = 'N', <i>u</i> is not referenced.
<i>v</i>	If <i>jobv</i> = 'I', <i>v</i> contains the orthogonal/unitary matrix <i>U</i> . If <i>jobv</i> = 'V', <i>v</i> contains the product $V_1 * V$. If <i>jobv</i> = 'N', <i>v</i> is not referenced.
<i>q</i>	If <i>jobq</i> = 'I', <i>q</i> contains the orthogonal/unitary matrix <i>U</i> . If <i>jobq</i> = 'Q', <i>q</i> contains the product $Q_1 * Q$. If <i>jobq</i> = 'N', <i>q</i> is not referenced.

ncycle

The number of cycles required for convergence.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, the procedure does not converge after MAXIT cycles.

Cosine-Sine Decomposition: LAPACK Computational Routines

This topic describes LAPACK computational routines for computing the *cosine-sine decomposition* (CS decomposition) of a partitioned unitary/orthogonal matrix. The algorithm computes a complete 2-by-2 CS decomposition, which requires simultaneous diagonalization of all the four blocks of a unitary/orthogonal matrix partitioned into a 2-by-2 block structure.

The computation has the following phases:

1. The matrix is reduced to a bidiagonal block form.
2. The blocks are simultaneously diagonalized using techniques from the bidiagonal SVD algorithms.

Table "Computational Routines for Cosine-Sine Decomposition (CSD)" lists LAPACK routines that perform CS decomposition of matrices.

Computational Routines for Cosine-Sine Decomposition (CSD)

Operation	Real matrices	Complex matrices
Compute the CS decomposition of an orthogonal/unitary matrix in bidiagonal-block form	bbcsd/bbcsd	bbcsd/bbcsd
Simultaneously bidiagonalize the blocks of a partitioned orthogonal matrix	orbdb unbdb	
Simultaneously bidiagonalize the blocks of a partitioned unitary matrix		orbdb unbdb

See Also

CS Driver Routine

?bbcsd

Computes the CS decomposition of an orthogonal/unitary matrix in bidiagonal-block form.

Syntax

```
lapack_int LAPACKE_sbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, float* theta, float* phi, float* u1, lapack_int ldu1, float* u2, lapack_int ldu2, float* v1t, lapack_int ldv1t, float* v2t, lapack_int ldv2t, float* b11d, float* b11e, float* b12d, float* b12e, float* b21d, float* b21e, float* b22d, float* b22e );
```

```
lapack_int LAPACKE_dbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, double* theta, double* phi, double* u1, lapack_int ldu1, double* u2, lapack_int ldu2, double* v1t, lapack_int ldv1t, double* v2t, lapack_int ldv2t, double* b11d, double* b11e, double* b12d, double* b12e, double* b21d, double* b21e, double* b22d, double* b22e );
```

```
lapack_int LAPACKE_cbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, float* theta, float* phi,
lapack_complex_float* u1, lapack_int ldu1, lapack_complex_float* u2, lapack_int ldu2,
lapack_complex_float* v1t, lapack_int ldv1t, lapack_complex_float* v2t, lapack_int
ldv2t, float* b11d, float* b11e, float* b12d, float* b12e, float* b21d, float* b21e,
float* b22d, float* b22e );
```

```
lapack_int LAPACKE_zbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, double* theta, double*
phi, lapack_complex_double* u1, lapack_int ldu1, lapack_complex_double* u2, lapack_int
ldu2, lapack_complex_double* v1t, lapack_int ldv1t, lapack_complex_double* v2t,
lapack_int ldv2t, double* b11d, double* b11e, double* b12d, double* b12e, double* b21d,
double* b21e, double* b22d, double* b22e );
```

Include Files

- mkl.h

Description

`mkl_lapack.fi` The routine `?bbcsd` computes the CS decomposition of an orthogonal or unitary matrix in bidiagonal-block form:

$$X = \begin{pmatrix} b_{11} & | & b_{12} & & 0 & 0 \\ 0 & | & 0 & & -I & 0 \\ b_{21} & | & b_{22} & & 0 & 0 \\ 0 & | & 0 & & 0 & I \end{pmatrix} = \begin{pmatrix} u_1 & | & \\ & & u_2 \end{pmatrix} \begin{pmatrix} C & | & -S & & 0 & 0 \\ 0 & | & 0 & & -I & 0 \\ S & | & C & & 0 & 0 \\ 0 & | & 0 & & 0 & I \end{pmatrix} \begin{pmatrix} v_1 & | & \\ & & v_2 \end{pmatrix}^T$$

or

$$X = \begin{pmatrix} b_{11} & | & b_{12} & & 0 & 0 \\ 0 & | & 0 & & -I & 0 \\ b_{21} & | & b_{22} & & 0 & 0 \\ 0 & | & 0 & & 0 & I \end{pmatrix} = \begin{pmatrix} u_1 & | & \\ & & u_2 \end{pmatrix} \begin{pmatrix} C & | & -S & & 0 & 0 \\ 0 & | & 0 & & -I & 0 \\ S & | & C & & 0 & 0 \\ 0 & | & 0 & & 0 & I \end{pmatrix} \begin{pmatrix} v_1 & | & \\ & & v_2 \end{pmatrix}^H$$

respectively.

x is m -by- m with the top-left block p -by- q . Note that q must not be larger than p , $m-p$, or $m-q$. If q is not the smallest index, x must be transposed and/or permuted in constant time using the `trans` option.

See `?orcsd`/`?uncsd` for details.

The bidiagonal matrices b_{11} , b_{12} , b_{21} , and b_{22} are represented implicitly by angles $theta(1:q)$ and $phi(1:q-1)$.

The orthogonal/unitary matrices u_1 , u_2 , v_1^t , and v_2^t are input/output. The input matrices are pre- or post-multiplied by the appropriate singular vector matrices.

Input Parameters

`matrix_layout` Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

`jobu1` If equals `Y`, then u_1 is updated. Otherwise, u_1 is not updated.

<i>jobu2</i>	If equals Y, then u_2 is updated. Otherwise, u_2 is not updated.
<i>jobv1t</i>	If equals Y, then v_1^t is updated. Otherwise, v_1^t is not updated.
<i>jobv2t</i>	If equals Y, then v_2^t is updated. Otherwise, v_2^t is not updated.
<i>trans</i>	= 'T': $x, u_1, u_2, v_1^t, v_2^t$ are stored in row-major order. otherwise $x, u_1, u_2, v_1^t, v_2^t$ are stored in column-major order.
<i>m</i>	The number of rows and columns of the orthogonal/unitary matrix X in bidiagonal-block form.
<i>p</i>	The number of rows in the top-left block of x . $0 \leq p \leq m$. \leq
<i>q</i>	The number of columns in the top-left block of x . $0 \leq q \leq \min(p, m-p, m-q)$.
<i>theta</i>	Array, size q . On entry, the angles $theta[0], \dots, theta[q-1]$ that, along with $phi[0], \dots, phi[q-2]$, define the matrix in bidiagonal-block form as returned by orbdb/unbdb .
<i>phi</i>	Array, size $q-1$. The angles $phi[0], \dots, phi[q-2]$ that, along with $theta[0], \dots, theta[q-1]$, define the matrix in bidiagonal-block form as returned by orbdb/unbdb .
<i>u1</i>	Array, size at least $\max(1, ldu1 \cdot p)$. On entry, a p -by- p matrix.
<i>ldu1</i>	The leading dimension of the array u_1 , $ldu1 \leq \max(1, p)$.
<i>u2</i>	Array, size $\max(1, ldu2 \cdot (m-p))$. On entry, an $(m-p)$ -by- $(m-p)$ matrix.
<i>ldu2</i>	The leading dimension of the array u_2 , $ldu2 \leq \max(1, m-p)$.
<i>v1t</i>	Array, size $\max(1, ldv1t \cdot q)$. On entry, a q -by- q matrix.
<i>ldv1t</i>	The leading dimension of the array $v1t$, $ldv1t \leq \max(1, q)$.
<i>v2t</i>	Array, size. On entry, an $(m-q)$ -by- $(m-q)$ matrix.
<i>ldv2t</i>	The leading dimension of the array $v2t$, $ldv2t \leq \max(1, m-q)$.

Output Parameters

<i>theta</i>	On exit, the angles whose cosines and sines define the diagonal blocks in the CS decomposition.
<i>u1</i>	On exit, $u1$ is postmultiplied by the left singular vector matrix common to $\begin{bmatrix} b11 & 0 \\ 0 & 0 \end{bmatrix}$ and $\begin{bmatrix} b12 & 0 & 0 \\ 0 & -I & 0 \end{bmatrix}$.

<i>u2</i>	On exit, <i>u2</i> is postmultiplied by the left singular vector matrix common to $\begin{bmatrix} b_{21} & 0 \end{bmatrix}$ and $\begin{bmatrix} b_{22} & 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.
<i>v1t</i>	Array, size <i>q</i> . On exit, <i>v1t</i> is premultiplied by the transpose of the right singular vector matrix common to $\begin{bmatrix} b_{11} & 0 \end{bmatrix}$ and $\begin{bmatrix} b_{21} & 0 \end{bmatrix}$.
<i>v2t</i>	On exit, <i>v2t</i> is premultiplied by the transpose of the right singular vector matrix common to $\begin{bmatrix} b_{12} & 0 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$ and $\begin{bmatrix} b_{22} & 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.
<i>b11d</i>	Array, size <i>q</i> . When ?bbcsd converges, <i>b11d</i> contains the cosines of <i>theta</i> [0], ..., <i>theta</i> [<i>q</i> - 1]. If ?bbcsd fails to converge, <i>b11d</i> contains the diagonal of the partially reduced top left block.
<i>b11e</i>	Array, size <i>q</i> -1. When ?bbcsd converges, <i>b11e</i> contains zeros. If ?bbcsd fails to converge, <i>b11e</i> contains the superdiagonal of the partially reduced top left block.
<i>b12d</i>	Array, size <i>q</i> . When ?bbcsd converges, <i>b12d</i> contains the negative sines of <i>theta</i> [0], ..., <i>theta</i> [<i>q</i> - 1]. If ?bbcsd fails to converge, <i>b12d</i> contains the diagonal of the partially reduced top right block.
<i>b12e</i>	Array, size <i>q</i> -1. When ?bbcsd converges, <i>b12e</i> contains zeros. If ?bbcsd fails to converge, <i>b11e</i> contains the superdiagonal of the partially reduced top right block.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0 and if ?bbcsd did not converge, *info* specifies the number of nonzero entries in *phi*, and *b11d*, *b11e*, etc. contain the partially reduced matrix.

See Also

[?orcscd/?uncscd](#)

[xerbla](#)

[?orbdb/?unbdb](#)

Simultaneously bidiagonalizes the blocks of a partitioned orthogonal/unitary matrix.

Syntax

```
lapack_int LAPACKE_sorbdb( int matrix_layout, char trans, char signs, lapack_int m,
lapack_int p, lapack_int q, float* x11, lapack_int ldx11, float* x12, lapack_int ldx12,
float* x21, lapack_int ldx21, float* x22, lapack_int ldx22, float* theta, float* phi,
float* taup1, float* taup2, float* tauq1, float* tauq2 );
```

```
lapack_int LAPACKE_dorbdb( int matrix_layout, char trans, char signs, lapack_int m,
lapack_int p, lapack_int q, double* x11, lapack_int ldx11, double* x12, lapack_int
ldx12, double* x21, lapack_int ldx21, double* x22, lapack_int ldx22, double* theta,
double* phi, double* taup1, double* taup2, double* tauq1, double* tauq );
```

```
lapack_int LAPACKE_cunbdb( int matrix_layout, char trans, char signs, lapack_int m,
lapack_int p, lapack_int q, lapack_complex_float* x11, lapack_int ldx11,
lapack_complex_float* x12, lapack_int ldx12, lapack_complex_float* x21, lapack_int
ldx21, lapack_complex_float* x22, lapack_int ldx22, float* theta, float* phi,
lapack_complex_float* taup1, lapack_complex_float* taup2, lapack_complex_float* tauq1,
lapack_complex_float* tauq2 );
```

```
lapack_int LAPACKE_zunbdb( int matrix_layout, char trans, char signs, lapack_int m,
lapack_int p, lapack_int q, lapack_complex_double* x11, lapack_int ldx11,
lapack_complex_double* x12, lapack_int ldx12, lapack_complex_double* x21, lapack_int
ldx21, lapack_complex_double* x22, lapack_int ldx22, double* theta, double* phi,
lapack_complex_double* taup1, lapack_complex_double* taup2, lapack_complex_double*
tauq1, lapack_complex_double* tauq2 );
```

Include Files

- mkl.h

Description

The routines ?orbdb/?unbdb simultaneously bidiagonalizes the blocks of an m -by- m partitioned orthogonal matrix X :

$$X = \begin{pmatrix} x_{11} & | & x_{12} \\ x_{21} & | & x_{22} \end{pmatrix} = \begin{pmatrix} p_1 & | \\ & p_2 \end{pmatrix} \begin{pmatrix} b_{11}|b_{12} & 0 & 0 \\ 0 & | & 0 & -I & 0 \\ b_{21}|b_{22} & 0 & 0 \\ 0 & | & 0 & 0 & I \end{pmatrix} \begin{pmatrix} q_1 & | \\ & q_2 \end{pmatrix}^T$$

or unitary matrix:

$$X = \begin{pmatrix} x_{11} & | & x_{12} \\ x_{21} & | & x_{22} \end{pmatrix} = \begin{pmatrix} p_1 & | \\ & p_2 \end{pmatrix} \begin{pmatrix} b_{11}|b_{12} & 0 & 0 \\ 0 & | & 0 & -I & 0 \\ b_{21}|b_{22} & 0 & 0 \\ 0 & | & 0 & 0 & I \end{pmatrix} \begin{pmatrix} q_1 & | \\ & q_2 \end{pmatrix}^H$$

x_{11} is p -by- q . q must not be larger than p , $m-p$, or $m-q$. Otherwise, x must be transposed and/or permuted in constant time using the *trans* and *signs* options.

The orthogonal/unitary matrices p_1 , p_2 , q_1 , and q_2 are p -by- p , $(m-p)$ -by- $(m-p)$, q -by- q , $(m-q)$ -by- $(m-q)$, respectively. They are represented implicitly by Householder vectors.

The bidiagonal matrices b_{11} , b_{12} , b_{21} , and b_{22} are q -by- q bidiagonal matrices represented implicitly by angles $theta[0], \dots, theta[q-1]$ and $phi[0], \dots, phi[q-2]$. b_{11} and b_{12} are upper bidiagonal, while b_{21} and b_{22} are lower bidiagonal. Every entry in each bidiagonal band is a product of a sine or cosine of $theta$ with a sine or cosine of phi . See [Sutton09] for details.

p_1 , p_2 , q_1 , and q_2 are represented as products of elementary reflectors. .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).	
<i>trans</i>	= 'T':	$x, u_1, u_2, v_1^t, v_2^t$ are stored in row-major order.
	otherwise	$x, u_1, u_2, v_1^t, v_2^t$ are stored in column-major order.
<i>signs</i>	= 'O':	The lower-left block is made nonpositive (the "other" convention).
	otherwise	The upper-right block is made nonpositive (the "default" convention).
<i>m</i>	The number of rows and columns of the matrix X .	
<i>p</i>	The number of rows in x_{11} and x_{12} . $0 \leq p \leq m$.	
<i>q</i>	The number of columns in x_{11} and x_{21} . $0 \leq q \leq \min(p, m-p, m-q)$.	
<i>x11</i>	Array, size (size $\max(1, ldx11*q)$ for column major layout and $\max(1, ldx11*p)$ for row major layout) . On entry, the top-left block of the orthogonal/unitary matrix to be reduced.	
<i>ldx11</i>	The leading dimension of the array X_{11} . If <i>trans</i> = 'T', $ldx11 \geq p$ for column major layout and $ldx11 \geq q$ for row major layout. Otherwise, $ldx11 \geq q$.	
<i>x12</i>	Array, size (size $\max(1, ldx12*(m-q))$ for column major layout and $\max(1, ldx12*p)$ for row major layout). On entry, the top-right block of the orthogonal/unitary matrix to be reduced.	
<i>ldx12</i>	The leading dimension of the array X_{12} . If <i>trans</i> = 'N', $ldx12 \geq p$ for column major layout and $ldx12 \geq m - q$ for row major layout. . Otherwise, $ldx12 \geq m - q$.	
<i>x21</i>	Array, size (size $\max(1, ldx21*q)$ for column major layout and $\max(1, ldx21*(m-p))$ for row major layout). On entry, the bottom-left block of the orthogonal/unitary matrix to be reduced.	
<i>ldx21</i>	The leading dimension of the array X_{21} . If <i>trans</i> = 'N', $ldx21 \geq m-p$ for column major layout and $ldx21 \geq q$ for row major layout. . Otherwise, $ldx21 \geq q$.	
<i>x22</i>	Array, size ((size $\max(1, ldx22*(m-q))$ for column major layout and $\max(1, ldx22*(m - p))$ for row major layout). On entry, the bottom-right block of the orthogonal/unitary matrix to be reduced.	
<i>ldx22</i>	The leading dimension of the array X_{21} . If <i>trans</i> = 'N', $ldx22 \geq m-p$ for column major layout and $ldx22 \geq m - q$ for row major layout. . Otherwise, $ldx22 \geq m - q$.	

Output Parameters

<i>x11</i>	<p>On exit, the form depends on <i>trans</i>:</p> <p>If <i>trans</i>='N', the columns of the lower triangle of <i>x11</i> specify reflectors for p_1, the rows of the upper triangle of <i>x11</i>(1:$q-1$, $q:q-1$) specify reflectors for q_1</p> <p>otherwise <i>trans</i>='T', the rows of the upper triangle of <i>x11</i> specify reflectors for p_1, the columns of the lower triangle of <i>x11</i>(1:$q-1$, $q:q-1$) specify reflectors for q_1</p>
<i>x12</i>	<p>On exit, the form depends on <i>trans</i>:</p> <p>If <i>trans</i>='N', the columns of the upper triangle of <i>x12</i> specify the first p reflectors for q_2</p> <p>otherwise <i>trans</i>='T', the columns of the lower triangle of <i>x12</i> specify the first p reflectors for q_2</p>
<i>x21</i>	<p>On exit, the form depends on <i>trans</i>:</p> <p>If <i>trans</i>='N', the columns of the lower triangle of <i>x21</i> specify the reflectors for p_2</p> <p>otherwise <i>trans</i>='T', the columns of the upper triangle of <i>x21</i> specify the reflectors for p_2</p>
<i>x22</i>	<p>On exit, the form depends on <i>trans</i>:</p> <p>If <i>trans</i>='N', the rows of the upper triangle of <i>x22</i>($q+1:m-p$, $p+1:m-q$) specify the last $m-p-q$ reflectors for q_2</p> <p>otherwise <i>trans</i>='T', the columns of the lower triangle of <i>x22</i>($p+1:m-q$, $+1:m-p$) specify the last $m-p-q$ reflectors for p_2</p>
<i>theta</i>	Array, size q . The entries of bidiagonal blocks b_{11} , b_{12} , b_{21} , and b_{22} can be computed from the angles <i>theta</i> and <i>phi</i> . See the Description section for details.
<i>phi</i>	Array, size $q-1$. The entries of bidiagonal blocks b_{11} , b_{12} , b_{21} , and b_{22} can be computed from the angles <i>theta</i> and <i>phi</i> . See the Description section for details.
<i>taup1</i>	Array, size p . Scalar factors of the elementary reflectors that define p_1 .
<i>taup2</i>	Array, size $m-p$. Scalar factors of the elementary reflectors that define p_2 .
<i>tauq1</i>	Array, size q . Scalar factors of the elementary reflectors that define q_1 .
<i>tauq2</i>	Array, size $m-q$. Scalar factors of the elementary reflectors that define q_2 .

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

See Also

[?orcscd/?uncscd](#)
[?orgqr](#)
[?ungqr](#)
[?orglq](#)
[?unglq](#)
[xerbla](#)

LAPACK Least Squares and Eigenvalue Problem Driver Routines

Each of the LAPACK driver routines solves a complete problem. To arrive at the solution, driver routines typically call a sequence of appropriate [computational routines](#).

Driver routines are described in the following topics :

[Linear Least Squares \(LLS\) Problems](#)

[Generalized LLS Problems](#)

[Symmetric Eigenproblems](#)

[Nonsymmetric Eigenproblems](#)

[Singular Value Decomposition](#)

[Cosine-Sine Decomposition](#)

[Generalized Symmetric Definite Eigenproblems](#)

[Generalized Nonsymmetric Eigenproblems](#)

Linear Least Squares (LLS) Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving linear least squares problems. [Table "Driver Routines for Solving LLS Problems"](#) lists all such routines.

Driver Routines for Solving LLS Problems

Routine Name	Operation performed
gels	Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.
gelsy	Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.
gelss	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.
gelsd	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

[?gels](#)

Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.

Syntax

```
lapack_int LAPACKE_sgels (int matrix_layout, char trans, lapack_int m, lapack_int n,
lapack_int nrhs, float* a, lapack_int lda, float* b, lapack_int ldb);
```

```

lapack_int LAPACKE_dgels (int matrix_layout, char trans, lapack_int m, lapack_int n,
lapack_int nrhs, double* a, lapack_int lda, double* b, lapack_int ldb);

lapack_int LAPACKE_cgels (int matrix_layout, char trans, lapack_int m, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b,
lapack_int ldb);

lapack_int LAPACKE_zgels (int matrix_layout, char trans, lapack_int m, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b,
lapack_int ldb);

```

Include Files

- mkl.h

Description

The routine solves overdetermined or underdetermined real/ complex linear systems involving an m -by- n matrix A , or its transpose/ conjugate-transpose, using a QR or LQ factorization of A . It is assumed that A has full rank.

The following options are provided:

1. If $trans = 'N'$ and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$$\text{minimize } ||b - A*x||_2$$

2. If $trans = 'N'$ and $m < n$: find the minimum norm solution of an underdetermined system $A^*X = B$.

3. If $trans = 'T'$ or $'C'$ and $m \geq n$: find the minimum norm solution of an undetermined system $A^H * X = B$.

4. If $trans = 'T'$ or $'C'$ and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$$\text{minimize } ||b - A^H * x||_2$$

Several right hand side vectors b and solution vectors x can be handled in a single call; they are formed by the columns of the right hand side matrix B and the solution matrix X (when coefficient matrix is A , B is m -by- $nrhs$ and X is n -by- $nrhs$; if the coefficient matrix is A^T or A^H , B is n -by- $nrhs$ and X is m -by- $nrhs$).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>trans</i>	Must be 'N', 'T', or 'C'. If $trans = 'N'$, the linear system involves matrix A ; If $trans = 'T'$, the linear system involves the transposed matrix A^T (for real flavors only); If $trans = 'C'$, the linear system involves the conjugate-transposed matrix A^H (for complex flavors only).
<i>m</i>	The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	The number of columns of the matrix A ($n \geq 0$).

<i>nrhs</i>	The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
<i>a, b</i>	Arrays: a (size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout) contains the m -by- n matrix A . b (size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*\max(m, n))$ for row major layout) contains the matrix B of right hand side vectors.
<i>lda</i>	The leading dimension of a ; at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.
<i>ldb</i>	The leading dimension of b ; must be at least $\max(1, m, n)$ for column major layout if $trans='N'$ and at least $\max(1, n)$ if $trans='T'$ and at least $\max(1, nrhs)$ for row major layout regardless of the value of $trans$.

Output Parameters

<i>a</i>	On exit, overwritten by the factorization data as follows: if $m \geq n$, array a contains the details of the QR factorization of the matrix A as returned by <code>?geqrf</code> ; if $m < n$, array a contains the details of the LQ factorization of the matrix A as returned by <code>?gelqf</code> .
<i>b</i>	If $info = 0$, b overwritten by the solution vectors, stored columnwise: if $trans = 'N'$ and $m \geq n$, rows 1 to n of b contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements $n+1$ to m in that column; if $trans = 'N'$ and $m < n$, rows 1 to n of b contain the minimum norm solution vectors; if $trans = 'T'$ or $'C'$ and $m \geq n$, rows 1 to m of b contain the minimum norm solution vectors; if $trans = 'T'$ or $'C'$ and $m < n$, rows 1 to m of b contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements $m+1$ to n in that column.

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of the triangular factor of A is zero, so that A does not have full rank; the least squares solution could not be computed.

?gelsy

Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A .

Syntax

```
lapack_int LAPACKE_sgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, float* a, lapack_int lda, float* b, lapack_int ldb, lapack_int* jpvt, float rcond,
lapack_int* rank );
```

```
lapack_int LAPACKE_dgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, double* a, lapack_int lda, double* b, lapack_int ldb, lapack_int* jpvt, double
rcond, lapack_int* rank );
```

```
lapack_int LAPACKE_cgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
lapack_int* jpvt, float rcond, lapack_int* rank );
```

```
lapack_int LAPACKE_zgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int
ldb, lapack_int* jpvt, double rcond, lapack_int* rank );
```

Include Files

- mkl.h

Description

The ?gelsy routine computes the minimum-norm solution to a real/complex linear least squares problem:

minimize $\|b - A \cdot x\|_2$

using a complete orthogonal factorization of A . A is an m -by- n matrix which may be rank-deficient. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- $nrhs$ right hand side matrix B and the n -by- $nrhs$ solution matrix X .

The routine first computes a QR factorization with column pivoting:

$$AP = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

with R_{11} defined as the largest leading submatrix whose estimated condition number is less than $1/rcond$. The order of R_{11} , $rank$, is the effective rank of A . Then, R_{22} is considered to be negligible, and R_{12} is annihilated by orthogonal/unitary transformations from the right, arriving at the complete orthogonal factorization:

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

The minimum-norm solution is then

$$X = PZ^T \begin{pmatrix} T_{11}^{-1} Q_1^T B \\ 0 \end{pmatrix}$$

for real flavors and

$$X = PZ^H \begin{pmatrix} T_{11}^{-1} Q_1^H B \\ 0 \end{pmatrix}$$

for complex flavors,

where Q_1 consists of the first *rank* columns of Q .

The `?gelsy` routine is identical to the original deprecated `?gelsx` routine except for the following differences:

- The call to the subroutine `?geqpf` has been substituted by the call to the subroutine `?geqp3`, which is a BLAS-3 version of the *QR* factorization with column pivoting.
- The matrix B (the right hand side) is updated with BLAS-3.
- The permutation of the matrix B (the right hand side) is faster and more simple.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>m</code>	The number of rows of the matrix A ($m \geq 0$).
<code>n</code>	The number of columns of the matrix A ($n \geq 0$).
<code>nrhs</code>	The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
<code>a, b</code>	Arrays: a (size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout) contains the m -by- n matrix A . b (size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*\max(m, n))$ for row major layout) contains the m -by- $nrhs$ right hand side matrix B .
<code>lda</code>	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

<i>ldb</i>	The leading dimension of <i>b</i> ; must be at least $\max(1, m, n)$ for column major layout and at least $\max(1, nrhs)$ for row major layout.
<i>jpvt</i>	Array, size at least $\max(1, n)$. On entry, if <i>jpvt</i> [<i>i</i> - 1] $\neq 0$, the <i>i</i> -th column of <i>A</i> is permuted to the front of <i>AP</i> , otherwise the <i>i</i> -th column of <i>A</i> is a free column.
<i>rcond</i>	<i>rcond</i> is used to determine the effective rank of <i>A</i> , which is defined as the order of the largest leading triangular submatrix R_{11} in the <i>QR</i> factorization with pivoting of <i>A</i> , whose estimated condition number $< 1/rcond$.

Output Parameters

<i>a</i>	On exit, overwritten by the details of the complete orthogonal factorization of <i>A</i> .
<i>b</i>	Overwritten by the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> .
<i>jpvt</i>	On exit, if <i>jpvt</i> [<i>i</i> - 1] = <i>k</i> , then the <i>i</i> -th column of <i>AP</i> was the <i>k</i> -th column of <i>A</i> .
<i>rank</i>	The effective rank of <i>A</i> , that is, the order of the submatrix R_{11} . This is the same as the order of the submatrix T_{11} in the complete orthogonal factorization of <i>A</i> .

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?gelss

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.

Syntax

```
lapack_int LAPACKGE_sgelss( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, float* a, lapack_int lda, float* b, lapack_int ldb, float* s, float rcond,
lapack_int* rank );
```

```
lapack_int LAPACKGE_dgelss( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, double* a, lapack_int lda, double* b, lapack_int ldb, double* s, double rcond,
lapack_int* rank );
```

```
lapack_int LAPACKGE_cgelss( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
float* s, float rcond, lapack_int* rank );
```

```
lapack_int LAPACKGE_zgelss( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int
ldb, double* s, double rcond, lapack_int* rank );
```

Include Files

- mkl.h

Description

The routine computes the minimum norm solution to a real linear least squares problem:

minimize $\|b - A*x\|_2$

using the singular value decomposition (SVD) of A . A is an m -by- n matrix which may be rank-deficient. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- $nrhs$ right hand side matrix B and the n -by- $nrhs$ solution matrix X . The effective rank of A is determined by treating as zero those singular values which are less than $rcond$ times the largest singular value.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	The number of columns of the matrix A ($n \geq 0$).
<i>nrhs</i>	The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
<i>a, b</i>	Arrays: a (size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout) contains the m -by- n matrix A . b (size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*\max(m, n))$ for row major layout) contains the m -by- $nrhs$ right hand side matrix B .
<i>lda</i>	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
<i>ldb</i>	The leading dimension of b ; must be at least $\max(1, m, n)$ for column major layout and at least $\max(1, nrhs)$ for row major layout.
<i>rcond</i>	$rcond$ is used to determine the effective rank of A . Singular values $s(i) \leq rcond * s(1)$ are treated as zero. If $rcond < 0$, machine precision is used instead.

Output Parameters

<i>a</i>	On exit, the first $\min(m, n)$ rows of a are overwritten with the matrix of right singular vectors of A , stored row-wise.
<i>b</i>	Overwritten by the n -by- $nrhs$ solution matrix X . If $m \geq n$ and $rank = n$, the residual sum-of-squares for the solution in the i -th column is given by the sum of squares of modulus of elements $n+1:m$ in that column.

<i>s</i>	Array, size at least $\max(1, \min(m, n))$. The singular values of <i>A</i> in decreasing order. The condition number of <i>A</i> in the 2-norm is $k_2(A) = s(1) / s(\min(m, n)) .$
<i>rank</i>	The effective rank of <i>A</i> , that is, the number of singular values which are greater than $rcond * s(1)$.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then the algorithm for computing the SVD failed to converge; *i* indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

?gelsd

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

Syntax

```
lapack_int LAPACKE_sgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, float* a, lapack_int lda, float* b, lapack_int ldb, float* s, float rcond,
lapack_int* rank );

lapack_int LAPACKE_dgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, double* a, lapack_int lda, double* b, lapack_int ldb, double* s, double rcond,
lapack_int* rank );

lapack_int LAPACKE_cgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
float* s, float rcond, lapack_int* rank );

lapack_int LAPACKE_zgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int
ldb, double* s, double rcond, lapack_int* rank );
```

Include Files

- mkl.h

Description

The routine computes the minimum-norm solution to a real linear least squares problem:

minimize $\|b - A*x\|_2$

using the singular value decomposition (SVD) of *A*. *A* is an *m*-by-*n* matrix which may be rank-deficient.

Several right hand side vectors *b* and solution vectors *x* can be handled in a single call; they are stored as the columns of the *m*-by-*nrhs* right hand side matrix *B* and the *n*-by-*nrhs* solution matrix *X*.

The problem is solved in three steps:

1. Reduce the coefficient matrix *A* to bidiagonal form with Householder transformations, reducing the original problem into a "bidiagonal least squares problem" (BLS).

2. Solve the BLS using a divide and conquer approach.
3. Apply back all the Householder transformations to solve the original least squares problem.

The effective rank of A is determined by treating as zero those singular values which are less than $rcond$ times the largest singular value.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>m</i>	The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	The number of columns of the matrix A ($n \geq 0$).
<i>nrhs</i>	The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
<i>a, b</i>	Arrays: a (size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout) contains the m -by- n matrix A . b (size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*\max(m, n))$ for row major layout) contains the m -by- $nrhs$ right hand side matrix B .
<i>lda</i>	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
<i>ldb</i>	The leading dimension of b ; must be at least $\max(1, m, n)$ for column major layout and at least $\max(1, nrhs)$ for row major layout.
<i>rcond</i>	$rcond$ is used to determine the effective rank of A . Singular values $s(i) \leq rcond * s(1)$ are treated as zero. If $rcond \leq 0$, machine precision is used instead.

Output Parameters

<i>a</i>	On exit, A has been overwritten.
<i>b</i>	Overwritten by the n -by- $nrhs$ solution matrix X . If $m \geq n$ and $rank = n$, the residual sum-of-squares for the solution in the i -th column is given by the sum of squares of modulus of elements $n+1:m$ in that column.
<i>s</i>	Array, size at least $\max(1, \min(m, n))$. The singular values of A in decreasing order. The condition number of A in the 2-norm is $k_2(A) = s(1) / s(\min(m, n))$.
<i>rank</i>	The effective rank of A , that is, the number of singular values which are greater than $rcond * s(1)$.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If $info = i$, then the algorithm for computing the SVD failed to converge; i indicates the number of off-diagonal elements of an intermediate bidiagonal form that did not converge to zero.

Generalized Linear Least Squares (LLS) Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving generalized linear least squares problems. [Table "Driver Routines for Solving Generalized LLS Problems"](#) lists all such routines.

Driver Routines for Solving Generalized LLS Problems

Routine Name	Operation performed
gglse	Solves the linear equality-constrained least squares problem using a generalized RQ factorization.
ggglm	Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

?gglse

Solves the linear equality-constrained least squares problem using a generalized RQ factorization.

Syntax

```
lapack_int LAPACKE_sgglse (int matrix_layout, lapack_int m, lapack_int n, lapack_int p,
float* a, lapack_int lda, float* b, lapack_int ldb, float* c, float* d, float* x);
```

```
lapack_int LAPACKE_dgglse (int matrix_layout, lapack_int m, lapack_int n, lapack_int p,
double* a, lapack_int lda, double* b, lapack_int ldb, double* c, double* d, double* x);
```

```
lapack_int LAPACKE_cgglse (int matrix_layout, lapack_int m, lapack_int n, lapack_int p,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* c, lapack_complex_float* d, lapack_complex_float* x);
```

```
lapack_int LAPACKE_zgglse (int matrix_layout, lapack_int m, lapack_int n, lapack_int p,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* c, lapack_complex_double* d, lapack_complex_double* x);
```

Include Files

- mkl.h

Description

The routine solves the linear equality-constrained least squares (LSE) problem:

minimize $\|c - A*x\|^2$ subject to $B*x = d$

where A is an m -by- n matrix, B is a p -by- n matrix, c is a given m -vector, and d is a given p -vector. It is assumed that $p \leq n \leq m+p$, and

$$\text{rank}(B) = p \quad \text{and} \quad \text{rank} \begin{pmatrix} A \\ B \end{pmatrix} = n .$$

These conditions ensure that the LSE problem has a unique solution, which is obtained using a generalized RQ factorization of the matrices (B, A) given by

$$B = (0 \ R)^* Q, \quad A = Z^* T^* Q$$

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>m</i>	The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	The number of columns of the matrices A and B ($n \geq 0$).
<i>p</i>	The number of rows of the matrix B ($0 \leq p \leq n \leq m+p$).
<i>a, b, c, d</i>	Arrays: a (size $\max(1, lda \cdot n)$ for column major layout and $\max(1, lda \cdot m)$ for row major layout) contains the m -by- n matrix A . b (size $\max(1, ldb \cdot n)$ for column major layout and $\max(1, ldb \cdot p)$ for row major layout) contains the p -by- n matrix B . c size at least $\max(1, m)$, contains the right hand side vector for the least squares part of the LSE problem. d , size at least $\max(1, p)$, contains the right hand side vector for the constrained equation.
<i>lda</i>	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
<i>ldb</i>	The leading dimension of b ; at least $\max(1, p)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	The elements on and above the diagonal contain the $\min(m, n)$ -by- n upper trapezoidal matrix T as returned by <code>?ggrqf</code> .
<i>x</i>	The solution of the LSE problem.
<i>b</i>	On exit, the upper right triangle contains the p -by- p upper triangular matrix R as returned by <code>?ggrqf</code> .
<i>d</i>	On exit, d is destroyed.
<i>c</i>	On exit, the residual sum-of-squares for the solution is given by the sum of squares of elements $n-p+1$ to m of vector c .

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

If $info = 1$, the upper triangular factor R associated with B in the generalized RQ factorization of the pair (B, A) is singular, so that $\text{rank}(B) < p$; the least squares solution could not be computed.

If $info = 2$, the $(n-p)$ -by- $(n-p)$ part of the upper trapezoidal factor T associated with A in the generalized RQ factorization of the pair (B, A) is singular, so that

$$\text{rank} \begin{pmatrix} A \\ B \end{pmatrix} < n$$

; the least squares solution could not be computed.

?sgglm

Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

Syntax

```
lapack_int LAPACKE_sggglm (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
float* a, lapack_int lda, float* b, lapack_int ldb, float* d, float* x, float* y);

lapack_int LAPACKE_dggglm (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
double* a, lapack_int lda, double* b, lapack_int ldb, double* d, double* x, double* y);

lapack_int LAPACKE_cggglm (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* d, lapack_complex_float* x, lapack_complex_float* y);

lapack_int LAPACKE_zggglm (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* d, lapack_complex_double* x, lapack_complex_double* y);
```

Include Files

- mkl.h

Description

The routine solves a general Gauss-Markov linear model (GLM) problem:

$\text{minimize}_x ||y||_2$ subject to $d = A*x + B*y$

where A is an n -by- m matrix, B is an n -by- p matrix, and d is a given n -vector. It is assumed that $m \leq n \leq m+p$, and $\text{rank}(A) = m$ and $\text{rank}(AB) = n$.

Under these assumptions, the constrained equation is always consistent, and there is a unique solution x and a minimal 2-norm solution y , which is obtained using a generalized QR factorization of the matrices (A, B) given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}; \quad B = Q * T * Z.$$

In particular, if matrix B is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

$$\text{minimize}_x ||B^{-1}(d-A*x)||_2.$$

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>n</code>	The number of rows of the matrices A and B ($n \geq 0$).
<code>m</code>	The number of columns in A ($m \geq 0$).
<code>p</code>	The number of columns in B ($p \geq n - m$).
<code>a, b, d</code>	Arrays: a (size $\max(1, lda*m)$ for column major layout and $\max(1, lda*n)$ for row major layout) contains the n -by- m matrix A . b (size $\max(1, ldb*p)$ for column major layout and $\max(1, ldb*n)$ for row major layout) contains the n -by- p matrix B . d , size at least $\max(1, n)$, contains the left hand side of the GLM equation.
<code>lda</code>	The leading dimension of a ; at least $\max(1, n)$ for column major layout and $\max(1, m)$ for row major layout.
<code>ldb</code>	The leading dimension of b ; at least $\max(1, n)$ for column major layout and $\max(1, p)$ for row major layout.

Output Parameters

<code>x, y</code>	Arrays x, y . size at least $\max(1, m)$ for x and at least $\max(1, p)$ for y . On exit, x and y are the solutions of the GLM problem.
<code>a</code>	On exit, the upper triangular part of the array a contains the m -by- m upper triangular matrix R .
<code>b</code>	On exit, if $n \leq p$, the upper right triangle contains the n -by- n upper triangular matrix T as returned by <code>?ggrqf</code> ; if $n > p$, the elements on and above the $(n-p)$ -th subdiagonal contain the n -by- p upper trapezoidal matrix T .
<code>d</code>	On exit, d is destroyed

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, the upper triangular factor *R* associated with *A* in the generalized QR factorization of the pair (*A*, *B*) is singular, so that $\text{rank}(A) < m$; the least squares solution could not be computed.

If *info* = 2, the bottom $(n-m)$ -by- $(n-m)$ part of the upper trapezoidal factor *T* associated with *B* in the generalized QR factorization of the pair (*A*, *B*) is singular, so that $\text{rank}(AB) < n$; the least squares solution could not be computed.

Symmetric Eigenvalue Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving symmetric eigenvalue problems. See also [computational routines](#) that can be called to solve these problems. [Table "Driver Routines for Solving Symmetric Eigenproblems"](#) lists all such driver routines.

Driver Routines for Solving Symmetric Eigenproblems

Routine Name	Operation performed
syev/heev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix.
syevd/heevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix using divide and conquer algorithm.
syevx/heevx	Computes selected eigenvalues and, optionally, eigenvectors of a symmetric / Hermitian matrix.
syevr/heevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix using the Relatively Robust Representations.
spev/hpev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
spevd/hpevd	Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix held in packed storage.
spevx/hpevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
sbev /hbev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
sbevd/hbevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian band matrix using divide and conquer algorithm.
sbevx/hbevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
stev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.
stevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.
stevx	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Routine Name	Operation performed
stevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

[?syev](#)

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix.

Syntax

```
lapack_int LAPACKE_ssyev (int matrix_layout, char jobz, char uplo, lapack_int n, float* a, lapack_int lda, float* w);
```

```
lapack_int LAPACKE_dsyev (int matrix_layout, char jobz, char uplo, lapack_int n, double* a, lapack_int lda, double* w);
```

Include Files

- mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A .

Note that for most cases of real symmetric eigenvalue problems the default choice should be [syevr](#) function as its underlying algorithm is faster and uses less workspace.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of A .
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>a</i>	<i>a</i> (size $\max(1, lda*n)$) is an array containing either upper or lower triangular part of the symmetric matrix A , as specified by <i>uplo</i> .
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit, if <i>jobz</i> = 'V', then if <i>info</i> = 0, array <i>a</i> contains the orthonormal eigenvectors of the matrix A . If <i>jobz</i> = 'N', then on exit the lower triangle
----------	---

(if `uplo = 'L'`) or the upper triangle (if `uplo = 'U'`) of A , including the diagonal, is overwritten.

`w`

Array, size at least $\max(1, n)$.

If `info = 0`, contains the eigenvalues of the matrix A in ascending order.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i`, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

?heev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
lapack_int LAPACKE_cheev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_float* a, lapack_int lda, float* w );
```

```
lapack_int LAPACKE_zheev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_double* a, lapack_int lda, double* w );
```

Include Files

- `mkl.h`

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A .

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be [heevr](#) function as its underlying algorithm is faster and uses less workspace.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobz</code>	Must be 'N' or 'V'. If <code>jobz = 'N'</code> , then only eigenvalues are computed. If <code>jobz = 'V'</code> , then eigenvalues and eigenvectors are computed.
<code>uplo</code>	Must be 'U' or 'L'. If <code>uplo = 'U'</code> , a stores the upper triangular part of A . If <code>uplo = 'L'</code> , a stores the lower triangular part of A .
<code>n</code>	The order of the matrix A ($n \geq 0$).

<i>a</i>	<i>a</i> (size $\max(1, lda \cdot n)$) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> .
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit, if <i>jobz</i> = 'V', then if <i>info</i> = 0, array <i>a</i> contains the orthonormal eigenvectors of the matrix <i>A</i> . If <i>jobz</i> = 'N', then on exit the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>w</i>	Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

?syevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric matrix using divide and conquer algorithm.

Syntax

```
lapack_int LAPACK_essyevd (int matrix_layout, char jobz, char uplo, lapack_int n,
float* a, lapack_int lda, float* w);

lapack_int LAPACK_dsyevd (int matrix_layout, char jobz, char uplo, lapack_int n,
double* a, lapack_int lda, double* w);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix *A*. In other words, it can compute the spectral factorization of *A* as: $A = Z \Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and *Z* is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Note that for most cases of real symmetric eigenvalue problems the default choice should be [syevr](#) function as its underlying algorithm is faster and uses less workspace. `?syevd` requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobz</code>	Must be 'N' or 'V'. If <code>jobz = 'N'</code> , then only eigenvalues are computed. If <code>jobz = 'V'</code> , then eigenvalues and eigenvectors are computed.
<code>uplo</code>	Must be 'U' or 'L'. If <code>uplo = 'U'</code> , <code>a</code> stores the upper triangular part of <code>A</code> . If <code>uplo = 'L'</code> , <code>a</code> stores the lower triangular part of <code>A</code> .
<code>n</code>	The order of the matrix <code>A</code> ($n \geq 0$).
<code>a</code>	Array, size (<code>lda</code> , *). <code>a</code> (size $\max(1, lda * n)$) is an array containing either upper or lower triangular part of the symmetric matrix <code>A</code> , as specified by <code>uplo</code> .
<code>lda</code>	The leading dimension of the array <code>a</code> . Must be at least $\max(1, n)$.

Output Parameters

<code>w</code>	Array, size at least $\max(1, n)$. If <code>info = 0</code> , contains the eigenvalues of the matrix <code>A</code> in ascending order. See also <code>info</code> .
<code>a</code>	If <code>jobz = 'V'</code> , then on exit this array is overwritten by the orthogonal matrix <code>Z</code> which contains the eigenvectors of <code>A</code> .

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = i`, and `jobz = 'N'`, then the algorithm failed to converge; `i` indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If `info = i`, and `jobz = 'V'`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns `info/(n+1)` through `mod(info, n+1)`.

If `info = -i`, the `i`-th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $\|E\|_2 = O(\varepsilon) * \|A\|_2$, where ε is the machine precision.

The complex analogue of this routine is [heevd](#)

?heevd

Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

Syntax

```
lapack_int LAPACKE_cheevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_float* a, lapack_int lda, float* w );
```

```
lapack_int LAPACKE_zheevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_double* a, lapack_int lda, double* w );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A . In other words, it can compute the spectral factorization of A as: $A = Z \Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be [heevr](#) function as its underlying algorithm is faster and uses less workspace. `?heevd` requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobz</code>	Must be 'N' or 'V'. If <code>jobz = 'N'</code> , then only eigenvalues are computed. If <code>jobz = 'V'</code> , then eigenvalues and eigenvectors are computed.
<code>uplo</code>	Must be 'U' or 'L'. If <code>uplo = 'U'</code> , <code>a</code> stores the upper triangular part of A . If <code>uplo = 'L'</code> , <code>a</code> stores the lower triangular part of A .
<code>n</code>	The order of the matrix A ($n \geq 0$).
<code>a</code>	<code>a</code> (size $\max(1, lda * n)$) is an array containing either upper or lower triangular part of the Hermitian matrix A , as specified by <code>uplo</code> .
<code>lda</code>	The leading dimension of the array <code>a</code> . Must be at least $\max(1, n)$.

Output Parameters

<i>w</i>	Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i> .
<i>a</i>	If <i>jobz</i> = 'V', then on exit this array is overwritten by the unitary matrix <i>Z</i> which contains the eigenvectors of <i>A</i> .

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = *i*, and *jobz* = 'N', then the algorithm failed to converge; *i* off-diagonal elements of an intermediate tridiagonal form did not converge to zero;

if *info* = *i*, and *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info*/(*n*+1) through *mod*(*info*, *n*+1).

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $\|E\|_2 = O(\varepsilon) * \|A\|_2$, where ε is the machine precision.

The real analogue of this routine is [syevd](#). See also [hpevd](#) for matrices held in packed storage, and [hbevd](#) for banded matrices.

?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

```
lapack_int LAPACKE_ssylvx (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, float* a, lapack_int lda, float vl, float vu, lapack_int il, lapack_int
iu, float abstol, lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* ifail);

lapack_int LAPACKE_dsylvx (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, double* a, lapack_int lda, double vl, double vu, lapack_int il, lapack_int
iu, double abstol, lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int*
ifail);
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of real symmetric eigenvalue problems the default choice should be [syevr](#) function as its underlying algorithm is faster and uses less workspace. *?syevx* is faster for a few selected eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	Must be 'A', 'V', or 'I'. If <i>range</i> = 'A', all eigenvalues will be found. If <i>range</i> = 'V', all eigenvalues in the half-open interval (<i>vl</i> , <i>vu</i>] will be found. If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	<i>a</i> (size $\max(1, lda * n)$) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> .
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>vl</i> , <i>vu</i>	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if <i>range</i> = 'A' or 'I'.
<i>il</i> , <i>iu</i>	If <i>range</i> = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints: $1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$, if $n = 0$. Not referenced if <i>range</i> = 'A' or 'V'.
<i>abstol</i>	The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$ for column major layout and $lda \geq \max(1, m)$ for row major layout.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>m</i>	The total number of eigenvalues found; $0 \leq m \leq n$.

If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu-il*+1.

w

Array, size at least $\max(1, n)$. The first *m* elements contain the selected eigenvalues of the matrix *A* in ascending order.

z

Array *z*(size $\max(1, ldz*m)$ for column major layout and $\max(1, ldz*n)$ for row major layout) contains eigenvectors.

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail

Array, size at least $\max(1, n)$.

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, then *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'V', then *ifail* is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon \cdot ||T||$ is used as tolerance, where $||T||$ is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{?lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * \text{?lamch}('S')$.

?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
lapack_int LAPACKE_cheevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_float* a, lapack_int lda, float vl, float vu, lapack_int
il, lapack_int iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z,
lapack_int ldz, lapack_int* ifail );
```

```
lapack_int LAPACKE_zheevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_double* a, lapack_int lda, double vl, double vu,
lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w,
lapack_complex_double* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be [heevr](#) function as its underlying algorithm is faster and uses less workspace. `?heevx` is faster for a few selected eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	Must be 'A', 'V', or 'I'. If <i>range</i> = 'A', all eigenvalues will be found. If <i>range</i> = 'V', all eigenvalues in the half-open interval (vl , vu] will be found. If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of A .
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>a</i>	<i>a</i> (size $\max(1, lda*n)$) is an array containing either upper or lower triangular part of the Hermitian matrix A , as specified by <i>uplo</i> .
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.

<i>vl, vu</i>	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if <i>range</i> = 'A' or 'I'.
<i>il, iu</i>	If <i>range</i> = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints: $1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$, if $n = 0$. Not referenced if <i>range</i> = 'A' or 'V'.
<i>abstol</i>	
<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$ for column major layout and $lda \geq \max(1, m)$ for row major layout.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>m</i>	The total number of eigenvalues found; $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.
<i>w</i>	Array, size $\max(1, n)$. The first <i>m</i> elements contain the selected eigenvalues of the matrix <i>A</i> in ascending order.
<i>z</i>	Array <i>z</i> (size $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout) contains eigenvectors. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>). If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>ifail</i>	Array, size at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, then <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'V', then <i>ifail</i> is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon \cdot ||T||$ will be used in its place, where $||T||$ is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 \cdot \text{?lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 \cdot \text{?lamch}('S')$.

?syevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using the Relatively Robust Representations.

Syntax

```
lapack_int LAPACKE_ssyeivr (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, float* a, lapack_int lda, float vl, float vu, lapack_int il, lapack_int
iu, float abstol, lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int*
isuppz);

lapack_int LAPACKE_dsyeivr (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, double* a, lapack_int lda, double vl, double vu, lapack_int il, lapack_int
iu, double abstol, lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int*
isuppz);
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix A to tridiagonal form T . Then, whenever possible, **?syevr** calls **stemr** to compute the eigenspectrum using Relatively Robust Representations. **stemr** computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L^*D^*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the each unreduced block of T :

- Compute $T - \sigma^*I = L^*D^*L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of D and L cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).
- For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine `?syevr` calls `stemr` when the full spectrum is requested on machines that conform to the IEEE-754 floating point standard. `?syevr` calls `stebz` and `stein` on non-IEEE machines and when partial spectrum requests are made.

Normal execution of `?dsyevr` may create NaNs and infinities and may abort due to a floating point exception in environments that do not handle NaNs and infinities in the IEEE standard default manner.

Note that `?syevr` is preferable for most cases of real symmetric eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu.$ If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> . For <i>range</i> = 'V' or 'I' and $iu-il < n-1$, <code>sstebz/dstebz</code> and <code>sstein/dstein</code> are called.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	<i>a</i> (size $\max(1, lda*n)$) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> .
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>vl, vu</i>	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$, if $n = 0$.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*.

If $abstol < n * \epsilon * ||T||$, then $n * \epsilon * ||T||$ is used instead, where ϵ is the machine precision, and $||T||$ is the 1-norm of the matrix *T*. The eigenvalues are computed to an accuracy of $\epsilon * ||T||$ irrespective of *abstol*.

If high relative accuracy is important, set *abstol* to `?lamch('S')`.

ldz

The leading dimension of the output array *z*.

Constraints:

$ldz \geq 1$ if *jobz* = 'N' and

$ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout if *jobz* = 'V'.

Output Parameters

a

On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m

The total number of eigenvalues found, $0 \leq m \leq n$.

If *range* = 'A', $m = n$, if *range* = 'I', $m = iu - il + 1$, and if *range* = 'V' the exact value of *m* is not known in advance.

w, *z*

Arrays:

w, size at least $\max(1, n)$, contains the selected eigenvalues in ascending order, stored in *w*[0] to *w*[*m* - 1];

z (size $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout) .

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*[*i* - 1].

If *jobz* = 'N', then *z* is not referenced.

isuppz

Array, size at least $2 * \max(1, m)$.

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*[2*i* - 2] through *isuppz*[2*i* - 1]. Referenced only if eigenvectors are needed (*jobz* = 'V') and all eigenvalues are needed, that is, *range* = 'A' or *range* = 'I' and *il* = 1 and *iu* = *n*.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i`, an internal error has occurred.

Application Notes

?heevr

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using the Relatively Robust Representations.

Syntax

```
lapack_int LAPACKCheevr( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_float* a, lapack_int lda, float vl, float vu, lapack_int
il, lapack_int iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z,
lapack_int ldz, lapack_int* isuppz );
```

```
lapack_int LAPACKZheevr( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_double* a, lapack_int lda, double vl, double vu,
lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w,
lapack_complex_double* z, lapack_int ldz, lapack_int* isuppz );
```

Include Files

- `mkl.h`

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix A to tridiagonal form T with a call to [hetrd](#). Then, whenever possible, *?heevr* calls [stegr](#) to compute the eigenspectrum using Relatively Robust Representations. *?stegr* computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L^*D^*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For each unreduced block (submatrix) of T :

- Compute $T - \sigma I = L^*D^*L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of D and L cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).
- For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine *?heevr* calls [stemr](#) when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard, or [stebz](#) and [stein](#) on non-IEEE machines and when partial spectrum requests are made.

Note that the routine `zheevr` is preferable for most cases of complex Hermitian eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobz</code>	Must be 'N' or 'V'. If <code>jobz = 'N'</code> , then only eigenvalues are computed. If <code>jobz = 'V'</code> , then eigenvalues and eigenvectors are computed.
<code>range</code>	Must be 'A' or 'V' or 'I'. If <code>range = 'A'</code> , the routine computes all eigenvalues. If <code>range = 'V'</code> , the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <code>range = 'I'</code> , the routine computes eigenvalues with indices <code>il</code> to <code>iu</code> . For <code>range = 'V'</code> or 'I', <code>sstebz/dstebz</code> and <code>cstein/zstein</code> are called.
<code>uplo</code>	Must be 'U' or 'L'. If <code>uplo = 'U'</code> , <code>a</code> stores the upper triangular part of <code>A</code> . If <code>uplo = 'L'</code> , <code>a</code> stores the lower triangular part of <code>A</code> .
<code>n</code>	The order of the matrix <code>A</code> ($n \geq 0$).
<code>a</code>	<code>a</code> (size $\max(1, lda * n)$) is an array containing either upper or lower triangular part of the Hermitian matrix <code>A</code> , as specified by <code>uplo</code> .
<code>lda</code>	The leading dimension of the array <code>a</code> . Must be at least $\max(1, n)$.
<code>vl, vu</code>	If <code>range = 'V'</code> , the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <code>range = 'A'</code> or 'I', <code>vl</code> and <code>vu</code> are not referenced.
<code>il, iu</code>	If <code>range = 'I'</code> , the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; <code>il</code> =1 and <code>iu</code> =0 if $n = 0$. If <code>range = 'A'</code> or 'V', <code>il</code> and <code>iu</code> are not referenced.
<code>abstol</code>	The absolute error tolerance to which each eigenvalue/eigenvector is required. If <code>jobz = 'V'</code> , the eigenvalues and eigenvectors output have residual norms bounded by <code>abstol</code> , and the dot products between different eigenvectors are bounded by <code>abstol</code> .

If $abstol < n * eps * ||T||$, then $n * eps * ||T||$ is used instead, where eps is the machine precision, and $||T||$ is the 1-norm of the matrix T . The eigenvalues are computed to an accuracy of $eps * ||T||$ irrespective of $abstol$.

If high relative accuracy is important, set $abstol$ to `?lamch('S')`.

ldz

The leading dimension of the output array z . Constraints:

$ldz \geq 1$ if $jobz = 'N'$;

$ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout if $jobz = 'V'$.

Output Parameters

a

On exit, the lower triangle (if $uplo = 'L'$) or the upper triangle (if $uplo = 'U'$) of A , including the diagonal, is overwritten.

m

The total number of eigenvalues found,

$0 \leq m \leq n$.

If $range = 'A'$, $m = n$, if $range = 'I'$, $m = iu - il + 1$, and if $range = 'V'$ the exact value of m is not known in advance.

w

Array, size at least $\max(1, n)$, contains the selected eigenvalues in ascending order, stored in $w[0]$ to $w[m - 1]$.

z

Array z (size $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout).

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w[i - 1]$.

If $jobz = 'N'$, then z is not referenced.

isuppz

Array, size at least $2 * \max(1, m)$.

The support of the eigenvectors in z , i.e., the indices indicating the nonzero elements in z . The i -th eigenvector is nonzero only in elements $isuppz[2i - 2]$ through $isuppz[2i - 1]$. Referenced only if eigenvectors are needed ($jobz = 'V'$) and all eigenvalues are needed, that is, $range = 'A'$ or $range = 'I'$ and $il = 1$ and $iu = n$.

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, an internal error has occurred.

Application Notes

Normal execution of `?stemr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

For more details, see [?stemr](#) and these references:

- Inderjit S. Dhillon and Beresford N. Parlett: "Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices," Linear Algebra and its Applications, 387(1), pp. 1-28, August 2004.
- Inderjit Dhillon and Beresford Parlett: "Orthogonal Eigenvectors and Relative Gaps," SIAM Journal on Matrix Analysis and Applications, Vol. 25, 2004. Also LAPACK Working Note 154.
- Inderjit Dhillon: "A new $O(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem", Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.

?spev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

```
lapack_int LAPACKE_sspev (int matrix_layout, char jobz, char uplo, lapack_int n, float* ap, float* w, float* z, lapack_int ldz);
```

```
lapack_int LAPACKE_dspev (int matrix_layout, char jobz, char uplo, lapack_int n, double* ap, double* w, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* in packed storage.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>ap</i>	Array <i>ap</i> contains the packed upper or lower triangle of symmetric matrix <i>A</i> , as specified by <i>uplo</i> . The size of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.
<i>ldz</i>	The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.

Output Parameters

w, z

Arrays:

w, size at least $\max(1, n)$.

If *info* = 0, *w* contains the eigenvalues of the matrix *A* in ascending order.

z (size $\max(1, ldz*n)$).

If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with *w*[*i* - 1].

If *jobz* = 'N', then *z* is not referenced.

ap

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

?hpev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

```
lapack_int LAPACKChpev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_float* ap, float* w, lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKzhpev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_double* ap, double* w, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* in packed storage.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

jobz

Must be 'N' or 'V'.

	If <code>job = 'N'</code> , then only eigenvalues are computed.
	If <code>job = 'V'</code> , then eigenvalues and eigenvectors are computed.
<code>uplo</code>	Must be 'U' or 'L'.
	If <code>uplo = 'U'</code> , <code>ap</code> stores the packed upper triangular part of <code>A</code> .
	If <code>uplo = 'L'</code> , <code>ap</code> stores the packed lower triangular part of <code>A</code> .
<code>n</code>	The order of the matrix <code>A</code> ($n \geq 0$).
<code>ap</code>	Array <code>ap</code> contains the packed upper or lower triangle of Hermitian matrix <code>A</code> , as specified by <code>uplo</code> .
	The size of <code>ap</code> must be at least $\max(1, n*(n+1)/2)$.
<code>ldz</code>	The leading dimension of the output array <code>z</code> .
	Constraints:
	if <code>jobz = 'N'</code> , then $ldz \geq 1$;
	if <code>jobz = 'V'</code> , then $ldz \geq \max(1, n)$.

Output Parameters

<code>w</code>	Array, size at least $\max(1, n)$.
	If <code>info = 0</code> , <code>w</code> contains the eigenvalues of the matrix <code>A</code> in ascending order.
<code>z</code>	Array <code>z</code> (size at least $\max(1, ldz*n)$).
	If <code>jobz = 'V'</code> , then if <code>info = 0</code> , <code>z</code> contains the orthonormal eigenvectors of the matrix <code>A</code> , with the i -th column of <code>z</code> holding the eigenvector associated with <code>w[i - 1]</code> .
	If <code>jobz = 'N'</code> , then <code>z</code> is not referenced.
<code>ap</code>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <code>A</code> .

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i`, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

?spevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix held in packed storage.

Syntax

```
lapack_int LAPACKE_sspevd (int matrix_layout, char jobz, char uplo, lapack_int n,
float* ap, float* w, float* z, lapack_int ldz);
```

```
lapack_int LAPACKE_dspevd (int matrix_layout, char jobz, char uplo, lapack_int n,
double* ap, double* w, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A (held in packed storage). In other words, it can compute the spectral factorization of A as:

$$A = Z \Lambda Z^T.$$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>ap</i>	<i>ap</i> contains the packed upper or lower triangle of symmetric matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be $\max(1, n*(n+1)/2)$
<i>ldz</i>	The leading dimension of the output array z . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.

Output Parameters

w, z

Arrays:

w, size at least $\max(1, n)$.

If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

z (size $\max(1, ldz*n)$).

If *jobz* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*. If *jobz* = 'N', then *z* is not referenced.

ap

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $\|E\|_2 = O(\varepsilon) * \|A\|_2$, where ε is the machine precision.

The complex analogue of this routine is [hpevd](#).

See also [syevd](#) for matrices held in full storage, and [sbevd](#) for banded matrices.

?hpevd

Uses divide and conquer algorithm to compute all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix held in packed storage.

Syntax

```
lapack_int LAPACKE_chpevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_float* ap, float* w, lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhpevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_double* ap, double* w, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix *A* (held in packed storage). In other words, it can compute the spectral factorization of *A* as: $A = Z * \Lambda * Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A * z_i = \lambda_i * z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>ap</i>	<i>ap</i> contains the packed upper or lower triangle of Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.
<i>ldz</i>	The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.

Output Parameters

<i>w</i>	Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i> .
<i>z</i>	Array, size 1 if <i>jobz</i> = 'N' and $\max(1, ldz*n)$ if <i>jobz</i> = 'V'. If <i>jobz</i> = 'V', then this array is overwritten by the unitary matrix <i>Z</i> which contains the eigenvectors of <i>A</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>ap</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <i>A</i> .

Return Values

This function returns a value *info*.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

The real analogue of this routine is [spevd](#).

See also [heevd](#) for matrices held in full storage, and [hbevd](#) for banded matrices.

?spevx

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

```
lapack_int LAPACKE_sspevx (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, float* ap, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* ifail);
```

```
lapack_int LAPACKE_dspevx (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, double* ap, double vl, double vu, lapack_int il, lapack_int iu, double
abstol, lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* ifail);
```

Include Files

- `mkl.h`

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A* in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobz</i>	Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .

<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of <i>A</i>.</p> <p>If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of <i>A</i>.</p>
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>ap</i>	<p>Array <i>ap</i> contains the packed upper or lower triangle of the symmetric matrix <i>A</i>, as specified by <i>uplo</i>.</p> <p>The size of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.</p>
<i>vl</i> , <i>vu</i>	<p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $vl < vu$.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il</i> , <i>iu</i>	<p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldz</i>	<p>The leading dimension of the output array <i>z</i>.</p> <p>Constraints:</p> <p>if <i>jobz</i> = 'N', then $ldz \geq 1$;</p> <p>if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout.</p>

Output Parameters

<i>ap</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <i>A</i> .
<i>m</i>	<p>The total number of eigenvalues found,</p> <p>$0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, if <i>range</i> = 'I', $m = iu - il + 1$, and if <i>range</i> = 'V' the exact value of <i>m</i> is not known in advance..</p>
<i>w</i> , <i>z</i>	<p>Arrays:</p> <p><i>w</i>, size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the selected eigenvalues of the matrix <i>A</i> in ascending order.</p> <p><i>z</i>(size $\max(1, ldz*m)$ for column major layout and $\max(1, ldz*n)$ for row major layout).</p>

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*[*i* - 1].

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

ifail

Array, size at least max(1, *n*).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon \|T\|_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * \text{lamch}('S')$.

?hpevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

```
lapack_int LAPACKE_chpevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_float* ap, float vl, float vu, lapack_int il, lapack_int
iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz,
lapack_int* ifail );
```

```
lapack_int LAPACKE_zhpevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_double* ap, double vl, double vu, lapack_int il,
lapack_int iu, double abstol, lapack_int* m, double* w, lapack_complex_double* z,
lapack_int ldz, lapack_int* ifail );
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>ap</i>	Array <i>ap</i> contains the packed upper or lower triangle of the Hermitian matrix A , as specified by <i>uplo</i> . The size of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.
<i>vl, vu</i>	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldz</i>	The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$;

if `jobz = 'V'`, then $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout.

Output Parameters

<code>ap</code>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of <i>A</i> .
<code>m</code>	The total number of eigenvalues found, $0 \leq m \leq n$. $0 \leq m \leq n$. If <code>range = 'A'</code> , $m = n$, if <code>range = 'I'</code> , $m = iu-il+1$, and if <code>range = 'V'</code> the exact value of <i>m</i> is not known in advance..
<code>w</code>	Array, size at least $\max(1, n)$. If <code>info = 0</code> , contains the selected eigenvalues of the matrix <i>A</i> in ascending order.
<code>z</code>	Array <i>z</i> (size $\max(1, ldz*m)$ for column major layout and $\max(1, ldz*n)$ for row major layout). If <code>jobz = 'V'</code> , then if <code>info = 0</code> , the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>). If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . If <code>jobz = 'N'</code> , then <i>z</i> is not referenced.
<code>ifail</code>	Array, size at least $\max(1, n)$. If <code>jobz = 'V'</code> , then if <code>info = 0</code> , the first <i>m</i> elements of <i>ifail</i> are zero; if <code>info > 0</code> , the <i>ifail</i> contains the indices the eigenvectors that failed to converge. If <code>jobz = 'N'</code> , then <i>ifail</i> is not referenced.

Return Values

This function returns a value *info*.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{?lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * \epsilon_{\text{mach}}('S')$.

?sbev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

```
lapack_int LAPACKE_ssbev (int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, float* ab, lapack_int ldab, float* w, float* z, lapack_int ldz);

lapack_int LAPACKE_dsbev (int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, double* ab, lapack_int ldab, double* w, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab</i>	<i>ab</i> (size at least $\max(1, ldab * n)$ for column major layout and at least $\max(1, ldab * (kd + 1))$ for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.
<i>ldab</i>	The leading dimension of <i>ab</i> ; must be at least <i>kd</i> + 1 for column major layout and <i>n</i> for row major layout.
<i>ldz</i>	The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$;

if `jobz = 'V'`, then $ldz \geq \max(1, n)$.

Output Parameters

`w, z`

Arrays:

`w`, size at least $\max(1, n)$.

If `info = 0`, contains the eigenvalues of the matrix *A* in ascending order.

`z`(size $\max(1, ldz*n)$).

If `jobz = 'V'`, then if `info = 0`, `z` contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of `z` holding the eigenvector associated with `w[i - 1]`.

If `jobz = 'N'`, then `z` is not referenced.

`ab`

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form (see the description of [?sbtrd](#)).

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

?hbev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

```
lapack_int LAPACKE_chbev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, lapack_complex_float* ab, lapack_int ldab, float* w,
lapack_complex_float* z, lapack_int ldz );
```

```
lapack_int LAPACKE_zhbev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, lapack_complex_double* ab, lapack_int ldab, double* w,
lapack_complex_double* z, lapack_int ldz );
```

Include Files

- `mkl.h`

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix *A*.

Input Parameters

`matrix_layout`

Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

`jobz`

Must be `'N'` or `'V'`.

If `jobz = 'N'`, then only eigenvalues are computed.

	If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab</i>	<i>ab</i> (size at least $\max(1, ldab*n)$ for column major layout and at least $\max(1, ldab*(kd + 1))$ for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.
<i>ldab</i>	The leading dimension of <i>ab</i> ; must be at least <i>kd</i> + 1 for column major layout and <i>n</i> for row major layout.
<i>ldz</i>	The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.

Output Parameters

<i>w</i>	Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	Array <i>z</i> (size $\max(1, ldz*n)$). If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the orthonormal eigenvectors of the matrix <i>A</i> , with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> [<i>i</i> - 1]. If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form(see the description of hbtrd).

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then the algorithm failed to converge;

i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

?sbevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric band matrix using divide and conquer algorithm.

Syntax

```
lapack_int LAPACKE_ssbevd (int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, float* ab, lapack_int ldab, float* w, float* z, lapack_int ldz);

lapack_int LAPACKE_dsbevd (int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, double* ab, lapack_int ldab, double* w, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric band matrix A . In other words, it can compute the spectral factorization of A as:

$$A = Z \Lambda Z^T$$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of A .
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>kd</i>	The number of super- or sub-diagonals in A ($kd \geq 0$).
<i>ab</i>	<i>ab</i> (size at least $\max(1, ldab \cdot n)$ for column major layout and at least $\max(1, ldab \cdot (kd + 1))$ for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by <i>uplo</i>) in band storage format.

ldab The leading dimension of *ab*; must be at least $kd+1$ for column major layout and n for row major layout.

ldz The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then $ldz \geq 1$;

if *jobz* = 'V', then $ldz \geq \max(1, n)$.

Output Parameters

w, z

Arrays:

w, size at least $\max(1, n)$.

If *info* = 0, contains the eigenvalues of the matrix *A* in ascending order. See also *info*.

z(size $\max(1, ldz*n$ if *job* = 'V' and at least 1 if *job* = 'N').

If *job* = 'V', then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*. The *i*-th column of *Z* contains the eigenvector which corresponds to the eigenvalue $w[i - 1]$.

If *job* = 'N', then *z* is not referenced.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = *i*, then the algorithm failed to converge; *i* indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

The complex analogue of this routine is [hbevd](#).

See also [syevd](#) for matrices held in full storage, and [spevd](#) for matrices held in packed storage.

?hbevd

Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian band matrix using divide and conquer algorithm.

Syntax

```
lapack_int LAPACKChbevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, lapack_complex_float* ab, lapack_int ldab, float* w,
lapack_complex_float* z, lapack_int ldz );
```

```
lapack_int LAPACKE_zhbevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, lapack_complex_double* ab, lapack_int ldab, double* w,
lapack_complex_double* z, lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian band matrix A . In other words, it can compute the spectral factorization of A as: $A = Z \Lambda Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A z_i = \lambda_i z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of A .
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>kd</i>	The number of super- or sub-diagonals in A ($kd \geq 0$).
<i>ab</i>	<i>ab</i> (size at least $\max(1, ldab \cdot n)$ for column major layout and at least $\max(1, ldab \cdot (kd + 1))$ for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by <i>uplo</i>) in band storage format.
<i>ldab</i>	The leading dimension of <i>ab</i> ; must be at least $kd+1$ for column major layout and n for row major layout.
<i>ldz</i>	The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then $ldz \geq 1$; if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.

Output Parameters

<i>w</i>	Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. See also <i>info</i> .
<i>z</i>	Array, size $\max(1, ldz*n)$ if <i>job</i> = 'V' and at least 1 if <i>job</i> = 'N'. If <i>jobz</i> = 'V', then this array is overwritten by the unitary matrix <i>Z</i> which contains the eigenvectors of <i>A</i> . The <i>i</i> -th column of <i>Z</i> contains the eigenvector which corresponds to the eigenvalue $w[i - 1]$. If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>ab</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $\|E\|_2 = O(\varepsilon) \|A\|_2$, where ε is the machine precision.

The real analogue of this routine is [sbevd](#).

See also [heevd](#) for matrices held in full storage, and [hpevd](#) for matrices held in packed storage.

*?sbev*x

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

```
lapack_int LAPACKE_ssbev(x, int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int kd, float* ab, lapack_int ldab, float* q, lapack_int ldq, float
vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w,
float* z, lapack_int ldz, lapack_int* ifail);
```

```
lapack_int LAPACKE_dsbev(x, int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int kd, double* ab, lapack_int ldab, double* q, lapack_int ldq,
double vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m,
double* w, double* z, lapack_int ldz, lapack_int* ifail);
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices in range <i>il</i> to <i>iu</i> .
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab</i>	Arrays: Array <i>ab</i> (size at least $\max(1, ldab*n)$ for column major layout and at least $\max(1, ldab*(kd + 1))$ for row major layout) contains either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.
<i>ldab</i>	The leading dimension of <i>ab</i> ; must be at least $kd + 1$ for column major layout and <i>n</i> for row major layout.
<i>vl, vu</i>	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldq, ldz</i>	The leading dimensions of the output arrays <i>q</i> and <i>z</i> , respectively.

Constraints:

$ldq \geq 1$, $ldz \geq 1$;

If $jobz = 'V'$, then $ldq \geq \max(1, n)$ and $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout .

Output Parameters

q

Array, size $\max(1, ldz * n)$.

If $jobz = 'V'$, the n -by- n orthogonal matrix is used in the reduction to tridiagonal form.

If $jobz = 'N'$, the array q is not referenced.

m

The total number of eigenvalues found, $0 \leq m \leq n$.

If $range = 'A'$, $m = n$, if $range = 'I'$, $m = iu - il + 1$, and if $range = 'V'$, the exact value of m is not known in advance.

w, z

Arrays:

w , size at least $\max(1, n)$. The first m elements of w contain the selected eigenvalues of the matrix A in ascending order.

z (size at least $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout).

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w[i - 1]$.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If $jobz = 'N'$, then z is not referenced.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

ifail

Array, size at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of *ifail* are zero; if $info > 0$, the *ifail* contains the indices the eigenvectors that failed to converge.

If $jobz = 'N'$, then *ifail* is not referenced.

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon \|T\|_1$ is used as tolerance, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{?lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 * \text{?lamch}('S')$.

?hbev

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

```
lapack_int LAPACKE_chbev( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int kd, lapack_complex_float* ab, lapack_int ldab,
lapack_complex_float* q, lapack_int ldq, float vl, float vu, lapack_int il, lapack_int
iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz,
lapack_int* ifail );
```

```
lapack_int LAPACKE_zhbev( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int kd, lapack_complex_double* ab, lapack_int ldab,
lapack_complex_double* q, lapack_int ldq, double vl, double vu, lapack_int il,
lapack_int iu, double abstol, lapack_int* m, double* w, lapack_complex_double* z,
lapack_int ldz, lapack_int* ifail );
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$.

	If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab</i>	<i>ab</i> (size at least $\max(1, ldab*n)$ for column major layout and at least $\max(1, ldab*(kd + 1))$ for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.
<i>ldab</i>	The leading dimension of <i>ab</i> ; must be at least <i>kd</i> + 1 for column major layout and <i>n</i> for row major layout.
<i>vl, vu</i>	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	The absolute error tolerance to which each eigenvalue is required. See <i>Application notes</i> for details on error tolerance.
<i>ldq, ldz</i>	The leading dimensions of the output arrays <i>q</i> and <i>z</i> , respectively. Constraints: $ldq \geq 1, ldz \geq 1$; If <i>jobz</i> = 'V', then $ldq \geq \max(1, n)$ and $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout.

Output Parameters

<i>q</i>	Array, size $\max(1, ldz*n)$. If <i>jobz</i> = 'V', the <i>n</i> -by- <i>n</i> unitary matrix is used in the reduction to tridiagonal form. If <i>jobz</i> = 'N', the array <i>q</i> is not referenced.
<i>m</i>	The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, if <i>range</i> = 'I', $m = iu - il + 1$, and if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance..

<i>w</i>	Array, size at least $\max(1, n)$. The first m elements contain the selected eigenvalues of the matrix A in ascending order.
<i>z</i>	<p>Array z (size at least $\max(1, ldz*m)$ for column major layout and $\max(1, ldz*n)$ for row major layout).</p> <p>If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of z holding the eigenvector associated with $w[i - 1]$.</p> <p>If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i>.</p> <p>If $jobz = 'N'$, then z is not referenced.</p>
<i>ab</i>	<p>On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.</p> <p>If $uplo = 'U'$, the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows kd and $kd+1$ of ab, and if $uplo = 'L'$, the diagonal and first subdiagonal of T are returned in the first two rows of ab.</p>
<i>ifail</i>	<p>Array, size at least $\max(1, n)$.</p> <p>If $jobz = 'V'$, then if $info = 0$, the first m elements of <i>ifail</i> are zero; if $info > 0$, the <i>ifail</i> contains the indices of the eigenvectors that failed to converge.</p> <p>If $jobz = 'N'$, then <i>ifail</i> is not referenced.</p>

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2*?lamch('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting *abstol* to $2*?lamch('S')$.

?stev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
lapack_int LAPACKE_sstev (int matrix_layout, char jobz, lapack_int n, float* d, float* e, float* z, lapack_int ldz);
```

```
lapack_int LAPACKE_dstev (int matrix_layout, char jobz, lapack_int n, double* d, double* e, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix *A*.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>d</i> , <i>e</i>	Arrays: Array <i>d</i> contains the <i>n</i> diagonal elements of the tridiagonal matrix <i>A</i> . The size of <i>d</i> must be at least $\max(1, n)$. Array <i>e</i> contains the <i>n</i> -1 subdiagonal elements of the tridiagonal matrix <i>A</i> . The size of <i>e</i> must be at least $\max(1, n)$. The <i>n</i> -th element of this array is used as workspace.
<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V' then $ldz \geq \max(1, n)$.

Output Parameters

<i>d</i>	On exit, if <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order.
<i>z</i>	Array, size (size $\max(1, ldz * n)$). If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the orthonormal eigenvectors of the matrix <i>A</i> , with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with the eigenvalue returned in <i>d</i> [<i>i</i> - 1]. If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>e</i>	On exit, this array is overwritten with intermediate results.

Return Values

This function returns a value *info*.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, then the algorithm failed to converge;

i elements of *e* did not converge to zero.

?stevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.

Syntax

```
lapack_int LAPACKESstevd (int matrix_layout, char jobz, lapack_int n, float* d, float* e, float* z, lapack_int ldz);
```

```
lapack_int LAPACKEdstevd (int matrix_layout, char jobz, lapack_int n, double* d, double* e, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix *T*. In other words, the routine can compute the spectral factorization of *T* as: $T = Z \Lambda Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and *Z* is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$T^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

There is no complex analogue of this routine.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>n</i>	The order of the matrix <i>T</i> ($n \geq 0$).
<i>d</i> , <i>e</i>	Arrays: <i>d</i> contains the <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i> . The dimension of <i>d</i> must be at least max(1, <i>n</i>). <i>e</i> contains the <i>n</i> -1 off-diagonal elements of <i>T</i> .

The dimension of e must be at least $\max(1, n)$. The n -th element of this array is used as workspace.

ldz

The leading dimension of the output array z . Constraints:

$ldz \geq 1$ if $job = 'N'$;

$ldz \geq \max(1, n)$ if $job = 'V'$.

Output Parameters

d

On exit, if $info = 0$, contains the eigenvalues of the matrix T in ascending order.

See also *info*.

z

Array, size $\max(1, ldz*n)$ if $jobz = 'V'$ and 1 if $jobz = 'N'$.

If $jobz = 'V'$, then this array is overwritten by the orthogonal matrix Z which contains the eigenvectors of T .

If $jobz = 'N'$, then z is not referenced.

e

On exit, this array is overwritten with intermediate results.

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If $info = -i$, the i -th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \epsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n) * \epsilon * \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$

Thus the accuracy of a computed eigenvector depends on the gap between its eigenvalue and all the other eigenvalues.

?stevx

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
lapack_int LAPACKE_sstevx (int matrix_layout, char jobz, char range, lapack_int n,
float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* ifail);
```

```
lapack_int LAPACKE_dstevx (int matrix_layout, char jobz, char range, lapack_int n,
double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu, double abstol,
lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* ifail);
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>d</i> , <i>e</i>	Arrays: <i>d</i> contains the n diagonal elements of the tridiagonal matrix A . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> contains the $n-1$ subdiagonal elements of A . The dimension of <i>e</i> must be at least $\max(1, n-1)$. The n -th element of this array is used as workspace.
<i>vl</i> , <i>vu</i>	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il</i> , <i>iu</i>	If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	

ldz The leading dimensions of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', then $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout.

Output Parameters

m The total number of eigenvalues found,
 $0 \leq m \leq n$.
 If *range* = 'A', $m = n$, if *range* = 'I', $m = iu - il + 1$, and if *range* = 'V' the exact value of *m* is unknown.

w, *z* Arrays:
w, size at least $\max(1, n)$.
 The first *m* elements of *w* contain the selected eigenvalues of the matrix *A* in ascending order.
z (size at least $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout) .
 If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with $w[i - 1]$.
 If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.
 If *jobz* = 'N', then *z* is not referenced.

d, *e* On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

ifail Array, size at least $\max(1, n)$.
 If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.
 If *jobz* = 'N', then *ifail* is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * |A|_1$ is used instead. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold $2 * ?lamch('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, set $abstol$ to $2 * \lambda_{\min}('S')$.

?stevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

Syntax

```
lapack_int LAPACKE_sstevr (int matrix_layout, char jobz, char range, lapack_int n,
float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* isuppz);

lapack_int LAPACKE_dstevr (int matrix_layout, char jobz, char range, lapack_int n,
double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu, double abstol,
lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* isuppz);
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, the routine calls [stemr](#) to compute the eigenspectrum using Relatively Robust Representations. [stegr](#) computes eigenvalues by the $dqds$ algorithm, while orthogonal eigenvectors are computed from various "good" $L * D * L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i -th unreduced block of T :

- Compute $T - \sigma_i = L_i * D_i * L_i^T$, such that $L_i * D_i * L_i^T$ is a relatively robust representation.
- Compute the eigenvalues, λ_j , of $L_i * D_i * L_i^T$ to high relative accuracy by the $dqds$ algorithm.
- If there is a cluster of close eigenvalues, "choose" σ_i close to the cluster, and go to Step (a).
- Given the approximate eigenvalue λ_j of $L_i * D_i * L_i^T$, compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter $abstol$.

The routine `?stevr` calls [stemr](#) when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. `?stevr` calls [stebz](#) and [stein](#) on non-IEEE machines and when partial spectrum requests are made.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If $jobz = 'N'$, then only eigenvalues are computed. If $jobz = 'V'$, then eigenvalues and eigenvectors are computed.
<i>range</i>	Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval:

$$vl < w[i] \leq vu.$$

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

For *range* = 'V' or 'I' and $iu - il < n - 1$, *sstebz/dstebz* and *sstein/dstein* are called.

n

The order of the matrix *T* ($n \geq 0$).

d, *e*

Arrays:

d contains the *n* diagonal elements of the tridiagonal matrix *T*.

The dimension of *d* must be at least $\max(1, n)$.

e contains the *n*-1 subdiagonal elements of *A*.

The dimension of *e* must be at least $\max(1, n-1)$. The *n*-th element of this array is used as workspace.

vl, *vu*

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu*

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

The absolute error tolerance to which each eigenvalue/eigenvector is required.

If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*. If $abstol < n * \text{eps} * ||T||$, then $n * \text{eps} * ||T||$ will be used in its place, where *eps* is the machine precision, and $||T||$ is the 1-norm of the matrix *T*. The eigenvalues are computed to an accuracy of $\text{eps} * ||T||$ irrespective of *abstol*.

If high relative accuracy is important, set *abstol* to `?lamch('S')`.

ldz

The leading dimension of the output array *z*.

Constraints:

$ldz \geq 1$ if *jobz* = 'N';

$ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout if *jobz* = 'V'.

Output Parameters

m

The total number of eigenvalues found,

$0 \leq m \leq n$. If *range* = 'A', $m = n$, if *range* = 'I', $m = iu - il + 1$, and if *range* = 'V' the exact value of *m* is unknown..

w, *z*

Arrays:

w, size at least $\max(1, n)$.

The first *m* elements of *w* contain the selected eigenvalues of the matrix *T* in ascending order.

z (size at least $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with $w[i - 1]$.

If *jobz* = 'N', then *z* is not referenced.

d, *e*

On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

isuppz

Array, size at least $2 * \max(1, m)$.

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*[2*i* - 2] through *isuppz*[2*i* - 1].

Implemented only for *range* = 'A' or 'I' and $iu - il = n - 1$.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, an internal error has occurred.

Application Notes

Normal execution of the routine ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

Nonsymmetric Eigenvalue Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving nonsymmetric eigenproblems. See also [computational routines](#) that can be called to solve these problems.

Table "Driver Routines for Solving Nonsymmetric Eigenproblems" lists all such driver routines.

Driver Routines for Solving Nonsymmetric Eigenproblems

Routine Name	Operation performed
gees	Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.
geesx	Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.

Routine Name	Operation performed
geev	Computes the eigenvalues and left and right eigenvectors of a general matrix.
geevx	Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

[?gees](#)

Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.

Syntax

```
lapack_int LAPACKE_sgees( int matrix_layout, char jobvs, char sort, LAPACK_S_SELECT2
select, lapack_int n, float* a, lapack_int lda, lapack_int* sdim, float* wr, float* wi,
float* vs, lapack_int ldvs );
```

```
lapack_int LAPACKE_dgees( int matrix_layout, char jobvs, char sort, LAPACK_D_SELECT2
select, lapack_int n, double* a, lapack_int lda, lapack_int* sdim, double* wr, double*
wi, double* vs, lapack_int ldvs );
```

```
lapack_int LAPACKE_cgees( int matrix_layout, char jobvs, char sort, LAPACK_C_SELECT1
select, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int* sdim,
lapack_complex_float* w, lapack_complex_float* vs, lapack_int ldvs );
```

```
lapack_int LAPACKE_zgees( int matrix_layout, char jobvs, char sort, LAPACK_Z_SELECT1
select, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_int* sdim,
lapack_complex_double* w, lapack_complex_double* vs, lapack_int ldvs );
```

Include Files

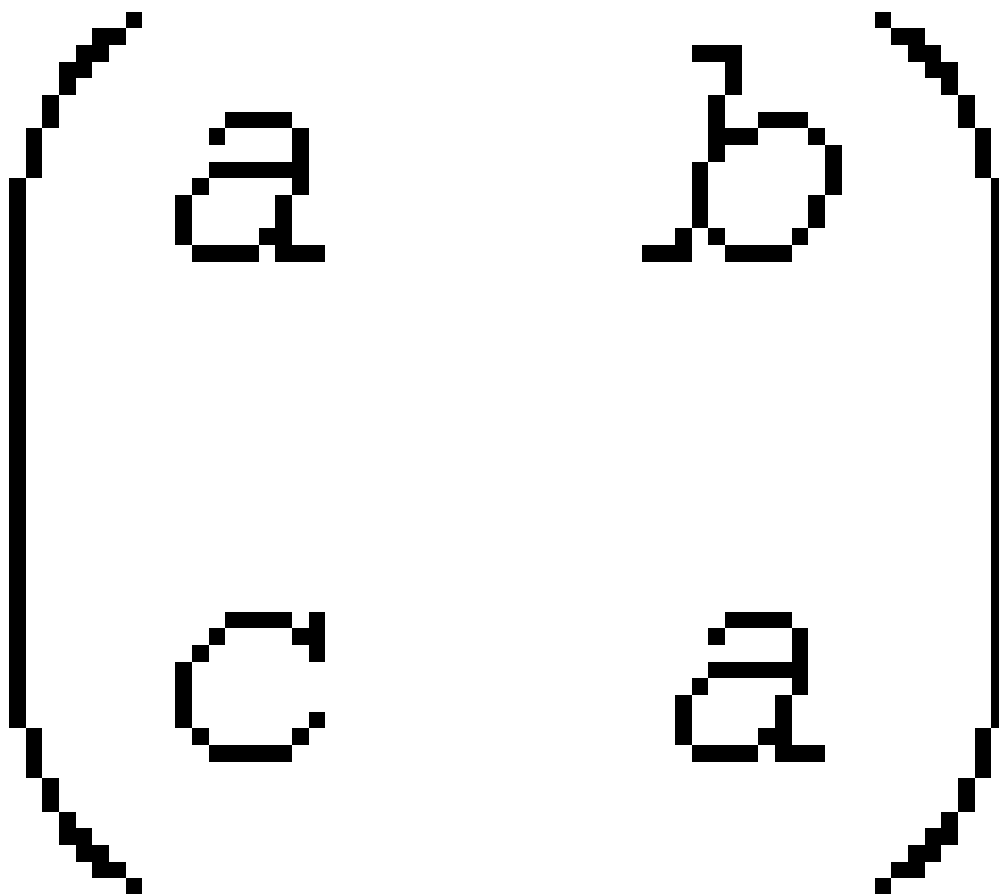
- `mk1.h`

Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues, the real Schur form T , and, optionally, the matrix of Schur vectors Z . This gives the Schur factorization $A = Z^* T^* Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left. The leading columns of Z then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form



where $b^*c < 0$. The eigenvalues of such a block are

$$a \pm i\sqrt{bc}$$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobvs</code>	Must be 'N' or 'V'. If <code>jobvs = 'N'</code> , then Schur vectors are not computed. If <code>jobvs = 'V'</code> , then Schur vectors are computed.
<code>sort</code>	Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. If <code>sort = 'N'</code> , then eigenvalues are not ordered. If <code>sort = 'S'</code> , eigenvalues are ordered (see <code>select</code>).
<code>select</code>	If <code>sort = 'S'</code> , <code>select</code> is used to select eigenvalues to sort to the top left of the Schur form.

If `sort = 'N'`, `select` is not referenced.

For real flavors:

An eigenvalue $wr[j] + \sqrt{-1} * wi[j]$ is selected if `select(wr[j], wi[j])` is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

For complex flavors:

An eigenvalue $w[j]$ is selected if `select(w[j])` is true.

Note that a selected complex eigenvalue may no longer satisfy `select(wr[j], wi[j]) = 1` after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case `info` may be set to $n+2$ (see *info* below).

`n` The order of the matrix A ($n \geq 0$).

`a` Arrays:

`a` (size at least $\max(1, lda * n)$) is an array containing the n -by- n matrix A .

`lda` The leading dimension of the array `a`. Must be at least $\max(1, n)$.

`ldvs` The leading dimension of the output array `vs`. Constraints:

$ldvs \geq 1$;

$ldvs \geq \max(1, n)$ if `jobvs = 'V'`.

Output Parameters

`a` On exit, this array is overwritten by the real-Schur/Schur form T .

`sdim` If `sort = 'N'`, `sdim` = 0.

If `sort = 'S'`, `sdim` is equal to the number of eigenvalues (after sorting) for which `select` is true.

Note that for real flavors complex conjugate pairs for which `select` is true for either eigenvalue count as 2.

`wr, wi` Arrays, size at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form T . Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

`w` Array, size at least $\max(1, n)$. Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form T .

`vs` Array `vs` (size at least $\max(1, ldvs * n)$).

If `jobvs = 'V'`, `vs` contains the orthogonal/unitary matrix Z of Schur vectors.

If `jobvs = 'N'`, `vs` is not referenced.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, and

`i ≤ n`:

the QR algorithm failed to compute all the eigenvalues; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged; if `jobvs = 'V'`, *vs* contains the matrix which reduces *A* to its partially converged Schur form;

`i = n+1`:

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

`i = n+2`:

after reordering, round-off changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy `select = 1`. This could also be caused by underflow due to scaling.

?geesx

Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.

Syntax

```
lapack_int LAPACKE_sgeesx( int matrix_layout, char jobvs, char sort, LAPACK_S_SELECT2
select, char sense, lapack_int n, float* a, lapack_int lda, lapack_int* sdim, float* wr,
float* wi, float* vs, lapack_int ldvs, float* rconde, float* rcondv );
```

```
lapack_int LAPACKE_dgeesx( int matrix_layout, char jobvs, char sort, LAPACK_D_SELECT2
select, char sense, lapack_int n, double* a, lapack_int lda, lapack_int* sdim, double*
wr, double* wi, double* vs, lapack_int ldvs, double* rconde, double* rcondv );
```

```
lapack_int LAPACKE_cgeesx( int matrix_layout, char jobvs, char sort, LAPACK_C_SELECT1
select, char sense, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int*
sdm, lapack_complex_float* w, lapack_complex_float* vs, lapack_int ldvs, float*
rconde, float* rcondv );
```

```
lapack_int LAPACKE_zgeesx( int matrix_layout, char jobvs, char sort, LAPACK_Z_SELECT1
select, char sense, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_int*
sdm, lapack_complex_double* w, lapack_complex_double* vs, lapack_int ldvs, double*
rconde, double* rcondv );
```

Include Files

- mkl.h

Description

The routine computes for an *n*-by-*n* real/complex nonsymmetric matrix *A*, the eigenvalues, the real-Schur/Schur form *T*, and, optionally, the matrix of Schur vectors *Z*. This gives the Schur factorization $A = Z^* T^* Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (*rconde*); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (*rcondv*). The leading columns of *Z* form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers $rconde$ and $rcondv$, see [LUG], Section 4.10 (where these quantities are called s and sep respectively).

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$

where $b*c < 0$. The eigenvalues of such a block are

$$a \pm i\sqrt{bc}$$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobvs</code>	Must be 'N' or 'V'. If <code>jobvs</code> = 'N', then Schur vectors are not computed. If <code>jobvs</code> = 'V', then Schur vectors are computed.
<code>sort</code>	Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.

	<p>If <code>sort = 'N'</code>, then eigenvalues are not ordered.</p> <p>If <code>sort = 'S'</code>, eigenvalues are ordered (see <i>select</i>).</p>
<i>select</i>	<p>If <code>sort = 'S'</code>, <i>select</i> is used to select eigenvalues to sort to the top left of the Schur form.</p> <p>If <code>sort = 'N'</code>, <i>select</i> is not referenced.</p> <p><i>For real flavors:</i></p> <p>An eigenvalue $wr[j] + \sqrt{-1} * wi[j]$ is selected if <i>select</i>($wr[j]$, $wi[j]$) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.</p> <p><i>For complex flavors:</i></p> <p>An eigenvalue $w[j]$ is selected if <i>select</i>($w[j]$) is true.</p> <p>Note that a selected complex eigenvalue may no longer satisfy <i>select</i>($wr[j]$, $wi[j]$) = 1 after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case <i>info</i> may be set to $n+2$ (see <i>info</i> below).</p>
<i>sense</i>	<p>Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.</p> <p>If <code>sense = 'N'</code>, none are computed;</p> <p>If <code>sense = 'E'</code>, computed for average of selected eigenvalues only;</p> <p>If <code>sense = 'V'</code>, computed for selected right invariant subspace only;</p> <p>If <code>sense = 'B'</code>, computed for both.</p> <p>If <i>sense</i> is 'E', 'V', or 'B', then <i>sort</i> must equal 'S'.</p>
<i>n</i>	The order of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	<p>Arrays:</p> <p><i>a</i> (size at least $\max(1, lda * n)$) is an array containing the n-by-n matrix <i>A</i>.</p>
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>ldvs</i>	<p>The leading dimension of the output array <i>vs</i>. Constraints:</p> <p>$ldvs \geq 1$;</p> <p>$ldvs \geq \max(1, n)$ if <i>jobvs</i> = 'V'.</p>

Output Parameters

<i>a</i>	On exit, this array is overwritten by the real-Schur/Schur form <i>T</i> .
<i>sdim</i>	<p>If <code>sort = 'N'</code>, <i>sdim</i> = 0.</p> <p>If <code>sort = 'S'</code>, <i>sdim</i> is equal to the number of eigenvalues (after sorting) for which <i>select</i> is true.</p> <p>Note that for real flavors complex conjugate pairs for which <i>select</i> is true for either eigenvalue count as 2.</p>

<i>wr, wi</i>	Arrays, size at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form T . Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.
<i>w</i>	Array, size at least $\max(1, n)$. Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form T .
<i>vs</i>	Array <i>vs</i> (size at least $\max(1, ldvs*n)$) If <i>jobvs</i> = 'V', <i>vs</i> contains the orthogonal/unitary matrix Z of Schur vectors. If <i>jobvs</i> = 'N', <i>vs</i> is not referenced.
<i>rconde, rcondv</i>	If <i>sense</i> = 'E' or 'B', <i>rconde</i> contains the reciprocal condition number for the average of the selected eigenvalues. If <i>sense</i> = 'N' or 'V', <i>rconde</i> is not referenced. If <i>sense</i> = 'V' or 'B', <i>rcondv</i> contains the reciprocal condition number for the selected right invariant subspace. If <i>sense</i> = 'N' or 'E', <i>rcondv</i> is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and

i ≤ *n*:

the QR algorithm failed to compute all the eigenvalues; elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain those eigenvalues which have converged; if *jobvs* = 'V', *vs* contains the transformation which reduces A to its partially converged Schur form;

i = *n*+1:

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

i = *n*+2:

after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy *select* = 1. This could also be caused by underflow due to scaling.

?geev

Computes the eigenvalues and left and right eigenvectors of a general matrix.

Syntax

```
lapack_int LAPACKE_sgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
float* a, lapack_int lda, float* wr, float* wi, float* vl, lapack_int ldvl, float* vr,
lapack_int ldvr );
```



```

lapack_int LAPACKE_dgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
double* a, lapack_int lda, double* wr, double* wi, double* vl, lapack_int ldvl, double*
vr, lapack_int ldvr );

lapack_int LAPACKE_cgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* w, lapack_complex_float*
vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr );

lapack_int LAPACKE_zgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* w,
lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int
ldvr );

```

Include Files

- mkl.h

Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues and, optionally, the left and/or right eigenvectors. The right eigenvector v of A satisfies

$$A*v = \lambda*v$$

where λ is its eigenvalue.

The left eigenvector u of A satisfies

$$u^H*A = \lambda*u^H$$

where u^H denotes the conjugate transpose of u . The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobvl</i>	Must be 'N' or 'V'. If <i>jobvl</i> = 'N', then left eigenvectors of A are not computed. If <i>jobvl</i> = 'V', then left eigenvectors of A are computed.
<i>jobvr</i>	Must be 'N' or 'V'. If <i>jobvr</i> = 'N', then right eigenvectors of A are not computed. If <i>jobvr</i> = 'V', then right eigenvectors of A are computed.
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>a</i>	a (size at least $\max(1, lda*n)$) is an array containing the n -by- n matrix A .
<i>lda</i>	The leading dimension of the array a . Must be at least $\max(1, n)$.
<i>ldvl, ldvr</i>	The leading dimensions of the output arrays vl and vr , respectively. Constraints: $ldvl \geq 1$; $ldvr \geq 1$. If <i>jobvl</i> = 'V', $ldvl \geq \max(1, n)$;

If $jobv_r = 'V'$, $ldv_r \geq \max(1, n)$.

Output Parameters

a

On exit, this array is overwritten.

w_r, w_i

Arrays, size at least $\max(1, n)$ each.

Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

Array, size at least $\max(1, n)$.

Contains the computed eigenvalues.

v_l, v_r

Arrays:

v_l (size at least $\max(1, ldv_l * n)$).

If $jobv_l = 'N'$, v_l is not referenced.

For real flavors:

If the j -th eigenvalue is real, the i -th component of the j -th eigenvector u_j is stored in $v_l[(i - 1) + (j - 1) * ldv_l]$ for column major layout and in $v_l[(i - 1) * ldv_l + (j - 1)]$ for row major layout..

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then for $i = \text{sqrt}(-1)$, the k -th component of the j -th eigenvector u_j is $v_l[(k - 1) + (j - 1) * ldv_l] + i * v_l[(k - 1) + j * ldv_l]$ for column major layout and as $v_l[(k - 1) * ldv_l + (j - 1)] + i * v_l[(k - 1) * ldv_l + j]$ for row major layout. Similarly, the k -th component of vector $(j+1)$ u_{j+1} is $v_l[(k - 1) + (j - 1) * ldv_l] - i * v_l[(k - 1) + j * ldv_l]$ for column major layout and as $v_l[(k - 1) * ldv_l + (j - 1)] - i * v_l[(k - 1) * ldv_l + j]$ for row major layout. .

For complex flavors:

The i -th component of the j -th eigenvector u_j is stored in $v_l[(i - 1) + (j - 1) * ldv_l]$ for column major layout and in $v_l[(i - 1) * ldv_l + (j - 1)]$ for row major layout.

v_r (size at least $\max(1, ldv_r * n)$).

If $jobv_r = 'N'$, v_r is not referenced.

For real flavors:

If the j -th eigenvalue is real, then the i -th component of j -th eigenvector v_j is stored in $v_r[(i - 1) + (j - 1) * ldv_r]$ for column major layout and in $v_r[(i - 1) * ldv_r + (j - 1)]$ for row major layout..

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then for $i = \text{sqrt}(-1)$, the k -th component of the j -th eigenvector v_j is $v_r[(k - 1) + (j - 1) * ldv_r] + i * v_r[(k - 1) + j * ldv_r]$ for column major layout and as $v_r[(k - 1) * ldv_r + (j - 1)] + i * v_r[(k - 1) * ldv_r + j]$ for row major layout. Similarly, the k -th component of vector $j + 1$ v_{j+1} is $v_r[(k - 1) + (j - 1) * ldv_r] - i * v_r[(k - 1) + j * ldv_r]$ for column major layout and as $v_r[(k - 1) * ldv_r + (j - 1)] - i * v_r[(k - 1) * ldv_r + j]$ for row major layout.

For complex flavors:

The i -th component of the j -th eigenvector v_j is stored in `vr[(i - 1) + (j - 1)*ldvr]` for column major layout and in `vr[(i - 1)*ldvr + (j - 1)]` for row major layout.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i`, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; elements $i+1:n$ of `wr` and `wi` (for real flavors) or `w` (for complex flavors) contain those eigenvalues which have converged.

?geevx

Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

Syntax

```
lapack_int LAPACKE_sgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, float* a, lapack_int lda, float* wr, float* wi, float* vl,
lapack_int ldvl, float* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, float*
scale, float* abnrm, float* rconde, float* rcondv );
```

```
lapack_int LAPACKE_dgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, double* a, lapack_int lda, double* wr, double* wi, double* vl,
lapack_int ldvl, double* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, double*
scale, double* abnrm, double* rconde, double* rcondv );
```

```
lapack_int LAPACKE_cgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* w,
lapack_complex_float* vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr,
lapack_int* ilo, lapack_int* ihi, float* scale, float* abnrm, float* rconde, float*
rcondv );
```

```
lapack_int LAPACKE_zgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double*
w, lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int
ldvr, lapack_int* ilo, lapack_int* ihi, double* scale, double* abnrm, double* rconde,
double* rcondv );
```

Include Files

- `mk1.h`

Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (`ilo`, `ihi`, `scale`, and `abnrm`), reciprocal condition numbers for the eigenvalues (`rconde`), and reciprocal condition numbers for the right eigenvectors (`rcondv`).

The right eigenvector v of A satisfies

$$A \cdot v = \lambda \cdot v$$

where λ is its eigenvalue.

The left eigenvector u of A satisfies

$$u^H A = \lambda u^H$$

where u^H denotes the conjugate transpose of u . The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation $D \cdot A \cdot \text{inv}(D)$, where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see [LUG], Section 4.10.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>balanc</i>	<p>Must be 'N', 'P', 'S', or 'B'. Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues.</p> <p>If <i>balanc</i> = 'N', do not diagonally scale or permute;</p> <p>If <i>balanc</i> = 'P', perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale;</p> <p>If <i>balanc</i> = 'S', diagonally scale the matrix, i.e. replace A by $D \cdot A \cdot \text{inv}(D)$, where D is a diagonal matrix chosen to make the rows and columns of A more equal in norm. Do not permute;</p> <p>If <i>balanc</i> = 'B', both diagonally scale and permute A.</p> <p>Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.</p>
<i>jobvl</i>	<p>Must be 'N' or 'V'.</p> <p>If <i>jobvl</i> = 'N', left eigenvectors of A are not computed;</p> <p>If <i>jobvl</i> = 'V', left eigenvectors of A are computed.</p> <p>If <i>sense</i> = 'E' or 'B', then <i>jobvl</i> must be 'V'.</p>
<i>jobvr</i>	<p>Must be 'N' or 'V'.</p> <p>If <i>jobvr</i> = 'N', right eigenvectors of A are not computed;</p> <p>If <i>jobvr</i> = 'V', right eigenvectors of A are computed.</p> <p>If <i>sense</i> = 'E' or 'B', then <i>jobvr</i> must be 'V'.</p>
<i>sense</i>	<p>Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.</p> <p>If <i>sense</i> = 'N', none are computed;</p>

If *sense* = 'E', computed for eigenvalues only;
 If *sense* = 'V', computed for right eigenvectors only;
 If *sense* = 'B', computed for eigenvalues and right eigenvectors.
 If *sense* is 'E' or 'B', both left and right eigenvectors must also be computed (*jobvl* = 'V' and *jobvr* = 'V').

n The order of the matrix *A* ($n \geq 0$).
a Arrays:
a (size at least $\max(1, lda*n)$) is an array containing the *n*-by-*n* matrix *A*.
lda The leading dimension of the array *a*. Must be at least $\max(1, n)$.
ldvl, ldvr The leading dimensions of the output arrays *vl* and *vr*, respectively.
 Constraints:
 $ldvl \geq 1; ldvr \geq 1$.
 If *jobvl* = 'V', $ldvl \geq \max(1, n)$;
 If *jobvr* = 'V', $ldvr \geq \max(1, n)$.

Output Parameters

a On exit, this array is overwritten.
 If *jobvl* = 'V' or *jobvr* = 'V', it contains the real-Schur/Schur form of the balanced version of the input matrix *A*.
wr, wi Arrays, size at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.
w Array, size at least $\max(1, n)$. Contains the computed eigenvalues.
vl, vr Arrays:
vl (size at least $\max(1, ldvl*n)$) .
 If *jobvl* = 'N', *vl* is not referenced.
For real flavors:
 If the *j*-th eigenvalue is real, the *i*-th component of the *j*-th eigenvector u_j is stored in $vl[(i - 1) + (j - 1)*ldvl]$ for column major layout and in $vl[(i - 1)*ldvl + (j - 1)]$ for row major layout..
 If the *j*-th and (*j*+1)-st eigenvalues form a complex conjugate pair, then for $i = \text{sqrt}(-1)$, the *k*-th component of the *j*-th eigenvector u_j is $vl[(k - 1) + (j - 1)*ldvl] + i*vl[(k - 1) + j*ldvl]$ for column major layout and as $vl[(k - 1)*ldvl + (j - 1)] + i*vl[(k-1)*ldvl + j]$ for row major layout. Similarly, the *k*-th component of vector (*j*+1) u_{j+1} is $vl[(k - 1) + (j - 1)*ldvl] - i*vl[(k - 1) + j*ldvl]$ for column major layout and as $vl[(k - 1)*ldvl + (j - 1)] - i*vl[(k - 1)*ldvl + j]$ for row major layout. .
For complex flavors:

The i -th component of the j -th eigenvector u_j is stored in $vl[(i - 1) + (j - 1)*ldvl]$ for column major layout and in $vl[(i - 1)*ldvl + (j - 1)]$ for row major layout.

vr (size at least $\max(1, ldvr*n)$).

If $jobvr = 'N'$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then the i -th component of j -th eigenvector v_j is stored in $vr[(i - 1) + (j - 1)*ldvr]$ for column major layout and in $vr[(i - 1)*ldvr + (j - 1)]$ for row major layout..

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then for $i = \text{sqrt}(-1)$, the k -th component of the j -th eigenvector v_j is $vr[(k - 1) + (j - 1)*ldvr] + i*vr[(k - 1) + j*ldvr]$ for column major layout and as $vr[(k - 1)*ldvr + (j - 1)] + i*vr[(k - 1)*ldvr + j]$ for row major layout. Similarly, the k -th component of vector $j + 1$ v_{j+1} is $vr[(k - 1) + (j - 1)*ldvr] - i*vr[(k - 1) + j*ldvr]$ for column major layout and as $vr[(k - 1)*ldvr + (j - 1)] - i*vr[(k - 1)*ldvr + j]$ for row major layout.

For complex flavors:

The i -th component of the j -th eigenvector v_j is stored in $vr[(i - 1) + (j - 1)*ldvr]$ for column major layout and in $vr[(i - 1)*ldvr + (j - 1)]$ for row major layout.

ilo, ihi

ilo and *ihi* are integer values determined when A was balanced.

The balanced $A(i, j) = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi + 1, \dots, n$.

If $balanc = 'N'$ or $'S'$, $ilo = 1$ and $ihi = n$.

scale

Array, size at least $\max(1, n)$. Details of the permutations and scaling factors applied when balancing A .

If $P[j - 1]$ is the index of the row and column interchanged with row and column j , and $D[j - 1]$ is the scaling factor applied to row and column j , then

$scale[j - 1] = P[j - 1]$, for $j = 1, \dots, ilo-1$
 $= D[j - 1]$, for $j = ilo, \dots, ihi$
 $= P[j - 1]$ for $j = ihi+1, \dots, n$.

The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

abnrm

The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

rconde, rcondv

Arrays, size at least $\max(1, n)$ each.

$rconde[j - 1]$ is the reciprocal condition number of the j -th eigenvalue.

$rcondv[j - 1]$ is the reciprocal condition number of the j -th right eigenvector.

Return Values

This function returns a value *info*.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors or condition numbers have been computed; elements 1://o-1 and *i*+1:*n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain eigenvalues which have converged.

Singular Value Decomposition: LAPACK Driver Routines

Table "Driver Routines for Singular Value Decomposition" lists the LAPACK driver routines that perform singular value decomposition .

Driver Routines for Singular Value Decomposition

Routine Name	Operation performed
?gesvd	Computes the singular value decomposition of a general rectangular matrix.
?gesdd	Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.
?gejsv	Computes the singular value decomposition of a real matrix using a preconditioned Jacobi SVD method.
?gesvj	Computes the singular value decomposition of a real matrix using Jacobi plane rotations.
?ggsvd	Computes the generalized singular value decomposition of a pair of general rectangular matrices.
?gesvdx	Computes the SVD and left and right singular vectors for a matrix.
?bdsvdx	Computes the SVD of a bidiagonal matrix.

Singular Value Decomposition - LAPACK Computational Routines

[?gesvd](#)

Computes the singular value decomposition of a general rectangular matrix.

Syntax

```
lapack_int LAPACKE_sgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
lapack_int n, float* a, lapack_int lda, float* s, float* u, lapack_int ldu, float* vt,
lapack_int ldvt, float* superb );
```

```
lapack_int LAPACKE_dgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
lapack_int n, double* a, lapack_int lda, double* s, double* u, lapack_int ldu, double*
vt, lapack_int ldvt, double* superb );
```

```
lapack_int LAPACKE_cgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
lapack_int n, lapack_complex_float* a, lapack_int lda, float* s, lapack_complex_float*
u, lapack_int ldu, lapack_complex_float* vt, lapack_int ldvt, float* superb );
```

```
lapack_int LAPACKE_zgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
lapack_int n, lapack_complex_double* a, lapack_int lda, double* s,
lapack_complex_double* u, lapack_int ldu, lapack_complex_double* vt, lapack_int ldvt,
double* superb );
```

Include Files

- `mk1.h`

Description

The routine computes the singular value decomposition (SVD) of a real/complex m -by- n matrix A , optionally computing the left and/or right singular vectors. The SVD is written as

$$A = U \Sigma V^T \text{ for real routines}$$

$$A = U \Sigma V^H \text{ for complex routines}$$

where Σ is an m -by- n matrix which is zero except for its $\min(m, n)$ diagonal elements, U is an m -by- m orthogonal/unitary matrix, and V is an n -by- n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A ; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of U and V are the left and right singular vectors of A .

The routine returns V^T (for real flavors) or V^H (for complex flavors), not V .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobu</i>	<p>Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix U.</p> <p>If <i>jobu</i> = 'A', all m columns of U are returned in the array u;</p> <p>if <i>jobu</i> = 'S', the first $\min(m, n)$ columns of U (the left singular vectors) are returned in the array u;</p> <p>if <i>jobu</i> = 'O', the first $\min(m, n)$ columns of U (the left singular vectors) are overwritten on the array a;</p> <p>if <i>jobu</i> = 'N', no columns of U (no left singular vectors) are computed.</p>
<i>jobvt</i>	<p>Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix V^T/V^H.</p> <p>If <i>jobvt</i> = 'A', all n rows of V^T/V^H are returned in the array vt;</p> <p>if <i>jobvt</i> = 'S', the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors) are returned in the array vt;</p> <p>if <i>jobvt</i> = 'O', the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors) are overwritten on the array a;</p> <p>if <i>jobvt</i> = 'N', no rows of V^T/V^H (no right singular vectors) are computed.</p> <p><i>jobvt</i> and <i>jobu</i> cannot both be 'O'.</p>
<i>m</i>	The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	The number of columns in A ($n \geq 0$).
<i>a</i>	<p>Arrays:</p> <p>a (size at least $\max(1, lda * n)$ for column major layout and $\max(1, lda * m)$ for row major layout) is an array containing the m-by-n matrix A.</p>
<i>lda</i>	<p>The leading dimension of the array a.</p> <p>Must be at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.</p>

$ldu, ldvt$

The leading dimensions of the output arrays u and vt , respectively.

Constraints:

$ldu \geq 1; ldvt \geq 1.$

If $jobu = 'A'$, $ldu \geq m$;

If $jobu = 'S'$, $ldu \geq m$ for column major layout and $ldu \geq \min(m, n)$ for row major layout;

If $jobvt = 'A'$, $ldvt \geq n$;

If $jobvt = 'S'$, $ldvt \geq \min(m, n)$ for column major layout and $ldvt \geq n$ for row major layout .

Output Parameters

 a

On exit,

If $jobu = 'O'$, a is overwritten with the first $\min(m, n)$ columns of U (the left singular vectors stored columnwise);

If $jobvt = 'O'$, a is overwritten with the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored rowwise);

If $jobu \neq 'O'$ and $jobvt \neq 'O'$, the contents of a are destroyed.

 s

Array, size at least $\max(1, \min(m, n))$. Contains the singular values of A sorted so that $s[i] \geq s[i + 1]$.

 u, vt

Arrays:

Array u minimum size:

	Column major layout	Row major layout
$jobu = 'A'$	$\max(1, ldu * m)$	$\max(1, ldu * m)$
$jobu = 'S'$	$\max(1, ldu * \min(m, n))$	$\max(1, ldu * m)$

If $jobu = 'A'$, u contains the m -by- m orthogonal/unitary matrix U .

If $jobu = 'S'$, u contains the first $\min(m, n)$ columns of U (the left singular vectors stored column-wise).

If $jobu = 'N'$ or $'O'$, u is not referenced.

Array v minimum size:

	Column major layout	Row major layout
$jobvt = 'A'$	$\max(1, ldvt * n)$	$\max(1, ldvt * n)$
$jobvt = 'S'$	$\max(1, ldvt * \min(m, n))$	$\max(1, ldvt * n)$

If $jobvt = 'A'$, vt contains the n -by- n orthogonal/unitary matrix V^T/V^H .

If $jobvt = 'S'$, vt contains the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored row-wise).

If `jobvt = 'N'` or `'O'`, `vt` is not referenced.

superb

If `?bdsqr` does not converge (indicated by the return value `info > 0`), on exit `superb(0:min(m,n)-2)` contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in `s` (not necessarily sorted). B satisfies $A = U^* B^* V^T$ (real flavors) or $A = U^* B^* V^H$ (complex flavors), so it has the same singular values as A , and singular vectors related by u and vt .

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i`, then if `?bdsqr` did not converge, i specifies how many superdiagonals of the intermediate bidiagonal form B did not converge to zero (see the description of the `superb` parameter for details).

?gesdd

Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.

Syntax

```
lapack_int LAPACKE_sgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
float* a, lapack_int lda, float* s, float* u, lapack_int ldu, float* vt, lapack_int
ldvt );
```

```
lapack_int LAPACKE_dgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
double* a, lapack_int lda, double* s, double* u, lapack_int ldu, double* vt, lapack_int
ldvt );
```

```
lapack_int LAPACKE_cgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, float* s, lapack_complex_float* u, lapack_int
ldu, lapack_complex_float* vt, lapack_int ldvt );
```

```
lapack_int LAPACKE_zgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, double* s, lapack_complex_double* u,
lapack_int ldu, lapack_complex_double* vt, lapack_int ldvt );
```

Include Files

- `mk1.h`

Description

The routine computes the singular value decomposition (SVD) of a real/complex m -by- n matrix A , optionally computing the left and/or right singular vectors.

If singular vectors are desired, it uses a divide-and-conquer algorithm. The SVD is written

$A = U^* \Sigma^* V^T$ for real routines,

$A = U^* \Sigma^* V^H$ for complex routines,

where Σ is an m -by- n matrix which is zero except for its $\min(m, n)$ diagonal elements, U is an m -by- m orthogonal/unitary matrix, and V is an n -by- n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A ; they are real and non-negative, and are returned in descending order. The first $\min(m, n)$ columns of U and V are the left and right singular vectors of A .

Note that the routine returns $vt = V^T$ (for real flavors) or $vt = V^H$ (for complex flavors), not V .

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

jobz Must be 'A', 'S', 'O', or 'N'.

Specifies options for computing all or part of the matrices U and V .

If *jobz* = 'A', all m columns of U and all n rows of V^T or V^H are returned in the arrays u and vt ;

if *jobz* = 'S', the first $\min(m, n)$ columns of U and the first $\min(m, n)$ rows of V^T or V^H are returned in the arrays u and vt ;

if *jobz* = 'O', then

if $m \geq n$, the first n columns of U are overwritten in the array a and all rows of V^T or V^H are returned in the array vt ;

if $m < n$, all columns of U are returned in the array u and the first m rows of V^T or V^H are overwritten in the array a ;

if *jobz* = 'N', no columns of U or rows of V^T or V^H are computed.

m The number of rows of the matrix A ($m \geq 0$).

n The number of columns in A ($n \geq 0$).

a $a(\text{size max}(1, lda*n))$ for column major layout and $\text{max}(1, lda*m)$ for row major layout) is an array containing the m -by- n matrix A .

lda The leading dimension of the array a . Must be at least $\text{max}(1, m)$ for column major layout and at least $\text{max}(1, n)$ for row major layout.

ldu, ldvt The leading dimensions of the output arrays u and vt , respectively.

The minimum size of *ldu* is

<i>jobz</i>	$m \geq n$	$m < n$
'N'	1	1
'A'	m	m
'S'	m for column major layout; n for row major layout	m
'O'	1	m

The minimum size of *ldvt* is

<i>jobz</i>	$m \geq n$	$m < n$
'N'	1	1

<i>jobz</i>	$m \geq n$	$m < n$
'A'	n	n
'S'	n	m for column major layout; n for row major layout
'O'	n	1

Output Parameters

a

On exit:

If *jobz* = 'O', then if $m \geq n$, *a* is overwritten with the first n columns of *U* (the left singular vectors, stored columnwise). If $m < n$, *a* is overwritten with the first m rows of V^T (the right singular vectors, stored rowwise);

If *jobz* ≠ 'O', the contents of *a* are destroyed.

s

Array, size at least $\max(1, \min(m, n))$. Contains the singular values of *A* sorted so that $s(i) \geq s(i+1)$.

u, vt

Arrays:

Array *u* is of size:

<i>jobz</i>	$m \geq n$	$m < n$
'N'	1	1
'A'	$\max(1, ldu * m)$	$\max(1, ldu * m)$
'S'	$\max(1, ldu * n)$ for column major layout; $\max(1, ldu * m)$ for row major layout	$\max(1, ldu * m)$
'O'	1	$\max(1, ldu * m)$

If *jobz* = 'A' or *jobz* = 'O' and $m < n$, *u* contains the m -by- m orthogonal/unitary matrix *U*.

If *jobz* = 'S', *u* contains the first $\min(m, n)$ columns of *U* (the left singular vectors, stored columnwise).

If *jobz* = 'O' and $m \geq n$, or *jobz* = 'N', *u* is not referenced.

Array *vt* is of size:

<i>jobz</i>	$m \geq n$	$m < n$
'N'	1	1
'A'	$\max(1, ldvt * n)$	$\max(1, ldvt * n)$
'S'	$\max(1, ldvt * n)$	$\max(1, ldvt * n)$ for column major layout; $\max(1, ldvt * m)$ for row major layout;

<i>jobz</i>	$m \geq n$	$m < n$
'O'	$\max(1, \text{ldvt} * n)$	1

If *jobz* = 'A' or *jobz* = 'O' and $m \geq n$, *vt* contains the *n*-by-*n* orthogonal/unitary matrix V^T .

If *jobz* = 'S', *vt* contains the first $\min(m, n)$ rows of V^T (the right singular vectors, stored rowwise).

If *jobz* = 'O' and $m < n$, or *jobz* = 'N', *vt* is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = -4, *A* had a NAN entry.

If *info* = *i*, then ?bdsdc did not converge, updating process failed.

?gejsv

Computes the singular value decomposition using a preconditioned Jacobi SVD method.

Syntax

```
lapack_int LAPACKGE_sgejsv (int matrix_layout, char joba, char jobu, char jobv, char
jobr, char jobt, char jobp, lapack_int m, lapack_int n, float * a, lapack_int lda, float
* sva, float * u, lapack_int ldu, float * v, lapack_int ldv, float * stat, lapack_int *
istat);
```

```
lapack_int LAPACKGE_dgejsv (int matrix_layout, char joba, char jobu, char jobv, char
jobr, char jobt, char jobp, lapack_int m, lapack_int n, double * a, lapack_int lda,
double * sva, double * u, lapack_int ldu, double * v, lapack_int ldv, double * stat,
lapack_int * istat);
```

```
lapack_int LAPACKGE_cgejsv (int matrix_layout, char joba, char jobu, char jobv, char
jobr, char jobt, char jobp, lapack_int m, lapack_int n, lapack_complex_float * a,
lapack_int lda, float * sva, lapack_complex_float * u, lapack_int ldu,
lapack_complex_float * v, lapack_int ldv, float * stat, lapack_int * istat);
```

```
lapack_int LAPACKGE_zgejsv (int matrix_layout, char joba, char jobu, char jobv, char
jobr, char jobt, char jobp, lapack_int m, lapack_int n, lapack_complex_double * a,
lapack_int lda, double * sva, lapack_complex_double * u, lapack_int ldu,
lapack_complex_double * v, lapack_int ldv, double * stat, lapack_int * istat);
```

Include Files

- mkl.h

Description

The routine computes the singular value decomposition (SVD) of a real/complex *m*-by-*n* matrix *A*, where $m \geq n$.

The SVD is written as

$A = U * \Sigma * V^T$, for real routines

$A = U \Sigma V^H$, for complex routines

where Σ is an m -by- n matrix which is zero except for its n diagonal elements, U is an m -by- n (or m -by- m) orthonormal matrix, and V is an n -by- n orthogonal matrix. The diagonal elements of Σ are the singular values of A ; the columns of U and V are the left and right singular vectors of A , respectively. The matrices U and V are computed and stored in the arrays u and v , respectively. The diagonal of Σ is computed and stored in the array sva .

The `?gejsv` routine can sometimes compute tiny singular values and their singular vectors much more accurately than other SVD routines.

The routine implements a preconditioned Jacobi SVD algorithm. It uses `?geqp3`, `?geqrf`, and `?gelqf` as preprocessors and preconditioners. Optionally, an additional row pivoting can be used as a preprocessor, which in some cases results in much higher accuracy. An example is matrix A with the structure $A = D1 * C * D2$, where $D1$, $D2$ are arbitrarily ill-conditioned diagonal matrices and C is a well-conditioned matrix. In that case, complete pivoting in the first QR factorizations provides accuracy dependent on the condition number of C , and independent of $D1$, $D2$. Such higher accuracy is not completely understood theoretically, but it works well in practice.

If A can be written as $A = B * D$, with well-conditioned B and some diagonal D , then the high accuracy is guaranteed, both theoretically and in software, independent of D . For more details see [Drmac08-1], [Drmac08-2].

The computational range for the singular values can be the full range (`UNDERFLOW,OVERFLOW`), provided that the machine arithmetic and the BLAS and LAPACK routines called by `?gejsv` are implemented to work in that range. If that is not the case, the restriction for safe computation with the singular values in the range of normalized IEEE numbers is that the spectral condition number $\kappa(A) = \sigma_{\max}(A) / \sigma_{\min}(A)$ does not overflow. This code (`?gejsv`) is best used in this restricted range, meaning that singular values of magnitude below $\|A\|_2 / \text{slamch}('O')$ (for single precision) or $\|A\|_2 / \text{dlamch}('O')$ (for double precision) are returned as zeros. See `jobr` for details on this.

This implementation is slower than the one described in [Drmac08-1], [Drmac08-2] due to replacement of some non-LAPACK components, and because the choice of some tuning parameters in the iterative part (`?gesvj`) is left to the implementer on a particular machine.

The rank revealing QR factorization (in this code: `?geqp3`) should be implemented as in [Drmac08-3].

If m is much larger than n , it is obvious that the initial QRF with column pivoting can be preprocessed by the QRF without pivoting. That well known trick is not used in `?gejsv` because in some cases heavy row weighting can be treated with complete pivoting. The overhead in cases m much larger than n is then only due to pivoting, but the benefits in accuracy have prevailed. You can incorporate this extra QRF step easily and also improve data movement (matrix transpose, matrix copy, matrix transposed copy) - this implementation of `?gejsv` uses only the simplest, naive data movement.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>joba</code>	Must be 'C', 'E', 'F', 'G', 'A', or 'R'.
	Specifies the level of accuracy:

If $joba = 'C'$, high relative accuracy is achieved if $A = B \cdot D$ with well-conditioned B and arbitrary diagonal matrix D . The accuracy cannot be spoiled by column scaling. The accuracy of the computed output depends on the condition of B , and the procedure aims at the best theoretical accuracy. The relative error $\max_{i=1:N} |d \sigma_i| / \sigma_i$ is bounded by $f(M,N) * \epsilon * \text{cond}(B)$, independent of D . The input matrix is preprocessed with the QRF with column pivoting. This initial preprocessing and preconditioning by a rank revealing QR factorization is common for all values of $joba$. Additional actions are specified as follows:

If $joba = 'E'$, computation as with 'C' with an additional estimate of the condition number of B . It provides a realistic error bound.

If $joba = 'F'$, accuracy higher than in the 'C' option is achieved, if $A = D1 \cdot C \cdot D2$ with ill-conditioned diagonal scalings $D1$, $D2$, and a well-conditioned matrix C . This option is advisable, if the structure of the input matrix is not known and relative accuracy is desirable. The input matrix A is preprocessed with QR factorization with full (row and column) pivoting.

If $joba = 'G'$, computation as with 'F' with an additional estimate of the condition number of B , where $A = B \cdot D$. If A has heavily weighted rows, using this condition number gives too pessimistic error bound.

If $joba = 'A'$, small singular values are the noise and the matrix is treated as numerically rank deficient. The error in the computed singular values is bounded by $f(m,n) * \epsilon * ||A||$. The computed SVD $A = U \cdot S \cdot V^T$ (for real flavors) or $A = U \cdot S \cdot V^H$ (for complex flavors) restores A up to $f(m,n) * \epsilon * ||A||$. This enables the procedure to set all singular values below $n * \epsilon * ||A||$ to zero.

If $joba = 'R'$, the procedure is similar to the 'A' option. Rank revealing property of the initial QR factorization is used to reveal (using triangular factor) a gap $\sigma_{r+1} < \epsilon * \sigma_r$, in which case the numerical rank is declared to be r . The SVD is computed with absolute error bounds, but more accurately than with 'A'.

Must be 'U', 'F', 'W', or 'N'.

Specifies whether to compute the columns of the matrix U :

If $jobu = 'U'$, n columns of U are returned in the array u

If $jobu = 'F'$, a full set of m left singular vectors is returned in the array u .

If $jobu = 'W'$, u may be used as workspace of length $m \cdot n$. See the description of u .

If $jobu = 'N'$, u is not computed.

Must be 'V', 'J', 'W', or 'N'.

Specifies whether to compute the matrix V :

If $jobv = 'V'$, n columns of V are returned in the array v ; Jacobi rotations are not explicitly accumulated.

If $jobv = 'J'$, n columns of V are returned in the array v but they are computed as the product of Jacobi rotations. This option is allowed only if $jobu \neq 'N'$

If `jobv = 'W'`, `v` may be used as workspace of length $n*n$. See the description of `v`.

If `jobv = 'N'`, `v` is not computed.

Must be 'N' or 'R'.

Specifies the range for the singular values. If small positive singular values are outside the specified range, they may be set to zero. If A is scaled so that the largest singular value of the scaled matrix is around `sqrt(big)`, `big = ?lamch('O')`, the function can remove columns of A whose norm in the scaled matrix is less than `sqrt(?lamch('S'))` (for `jobr = 'R'`), or less than `small = ?lamch('S')/?lamch('E')`.

If `jobr = 'N'`, the function does not remove small columns of the scaled matrix. This option assumes that BLAS and QR factorizations and triangular solvers are implemented to work in that range. If the condition of A is greater than `big`, use `?gesvj`.

If `jobr = 'R'`, restricted range for singular values of the scaled matrix A is `[sqrt(?lamch('S')), sqrt(big)]`, roughly as described above. This option is recommended.

For computing the singular values in the full range `[?lamch('S'), big]`, use `?gesvj`.

Must be 'T' or 'N'.

If the matrix is square, the procedure may determine to use a transposed A if A^T (for real flavors) or A^H (for complex flavors) seems to be better with respect to convergence. If the matrix is not square, `jobt` is ignored.

The decision is based on two values of entropy over the adjoint orbit of $A^T * A$ (for real flavors) or $A^H * A$ (for complex flavors). See the descriptions of `stat[5]` and `stat[6]`.

If `jobt = 'T'`, the function performs transposition if the entropy test indicates possibly faster convergence of the Jacobi process, if A is taken as input. If A is replaced with A^T or A^H , the row pivoting is included automatically.

If `jobt = 'N'`, the functions attempts no speculations. This option can be used to compute only the singular values, or the full SVD (u , σ , and v). For only one set of singular vectors (u or v), the caller should provide both u and v , as one of the arrays is used as workspace if the matrix A is transposed. The implementer can easily remove this constraint and make the code more complicated. See the descriptions of u and v .

Caution

The `jobt = 'T'` option is experimental and its effect might not be the same in subsequent releases. Consider using the `jobt = 'N'` instead.

Must be 'P' or 'N'.

Enables structured perturbations of denormalized numbers. This option should be active if the denormals are poorly implemented, causing slow computation, especially in cases of fast convergence. For details, see [Drmac08-1], [Drmac08-2]. For simplicity, such perturbations are included only when the full SVD or only the singular values are requested. You can add the perturbation for the cases of computing one set of singular vectors.

If `jobp = 'P'`, the function introduces perturbation.

If `jobp = 'N'`, the function introduces no perturbation.

`m`

The number of rows of the input matrix *A*; $m \geq 0$.

`n`

The number of columns in the input matrix *A*; $m \geq n \geq 0$.

`a, u, v`

Array *a* (size $lda*n$ for column major layout and $lda*m$ for row major layout) is an array containing the *m*-by-*n* matrix *A*.

u is a workspace array, its size for column major layout is $ldu*n$ for `jobu='U'` or `'W'` and $ldu*m$ for `jobu='F'`; for row major layout its size is at least $ldu*m$. When `jobt = 'T'` and $m = n$, *u* must be provided even though `jobu = 'N'`.

v is a workspace array, its size is $ldv*n$. When `jobt = 'T'` and $m = n$, *v* must be provided even though `jobv = 'N'`.

`lda`

The leading dimension of the array *a*. Must be at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.

`sva`

sva is a workspace array, its size is *n*.

`ldu`

The leading dimension of the array *u*; $ldu \geq 1$.

`jobu = 'U'` or `'F'` or `'W'`, $ldu \geq m$ for column major layout; for row major layout if `jobu = 'U'` or `jobu = 'W'` $ldu \geq n$ and if `jobu = 'F'` $ldu \geq m$.

`ldv`

The leading dimension of the array *v*; $ldv \geq 1$.

`jobv = 'V'` or `'J'` or `'W'`, $ldv \geq n$.

`cwork`

cwork is a workspace array of size $\max(2, lwork)$.

`rwork`

rwork is an array of size at least $\max(7, lrwork)$ for real flavors and at least $\max(7, lwork)$ for complex flavors.

Output Parameters

`sva`

On exit:

For `stat[0]/stat[1] = 1`: the singular values of *A*. During the computation *sva* contains Euclidean column norms of the iterated matrices in the array *a*.

For `stat[0] ≠ stat[1]`: the singular values of *A* are $(stat[0]/stat[1]) * sva[0:n - 1]$. This factored form is used if `sigma_max(A)` overflows or if small singular values have been saved from underflow by scaling the input matrix *A*.

$jobr = 'R'$, some of the singular values may be returned as exact zeros obtained by 'setting to zero' because they are below the numerical rank threshold or are denormalized numbers.

u

On exit:

If $jobu = 'U'$, contains the m -by- n matrix of the left singular vectors.

If $jobu = 'F'$, contains the m -by- m matrix of the left singular vectors, including an orthonormal basis of the orthogonal complement of the range of A .

If $jobu = 'W'$ and $jobv = 'V'$, $jobt = 'T'$, and $m = n$, then u is used as workspace if the procedure replaces A with A^T (for real flavors) or A^H (for complex flavors). In that case, v is computed in u as left singular vectors of A^T or A^H and copied back to the v array. This 'W' option is just a reminder to the caller that in this case u is reserved as workspace of length $n*n$.

If $jobu = 'N'$, u is not referenced.

v

On exit:

If $jobv = 'V'$ or $'J'$, contains the n -by- n matrix of the right singular vectors.

If $jobv = 'W'$ and $jobu = 'U'$, $jobt = 'T'$, and $m = n$, then v is used as workspace if the procedure replaces A with A^T (for real flavors) or A^H (for complex flavors). In that case, u is computed in v as right singular vectors of A^T or A^H and copied back to the u array. This 'W' option is just a reminder to the caller that in this case v is reserved as workspace of length $n*n$.

If $jobv = 'N'$, v is not referenced.

$stat$

On exit,

$stat[0] = scale = stat[1]/stat[0]$ is the scaling factor such that $scale*sva(1:n)$ are the computed singular values of A . See the description of sva .

$stat[1] =$ see the description of $stat[0]$.

$stat[2] = sconda$ is an estimate for the condition number of column equilibrated A . If $joba = 'E'$ or $'G'$, $sconda$ is an estimate of $\sqrt{|| (R^T * R)^{-1} ||_1}$. It is computed using `?pocon`. It holds $n^{-1/4} * sconda \leq || R^{-1} ||_2 \leq n^{1/4} * sconda$, where R is the triangular factor from the QRF of A . However, if R is truncated and the numerical rank is determined to be strictly smaller than n , $sconda$ is returned as -1, indicating that the smallest singular values might be lost.

If full SVD is needed, the following two condition numbers are useful for the analysis of the algorithm. They are provided for a user who is familiar with the details of the method.

$stat[3] =$ an estimate of the scaled condition number of the triangular factor in the first QR factorization.

$stat[4] =$ an estimate of the scaled condition number of the triangular factor in the second QR factorization.

The following two parameters are computed if *jobt* = 'T'. They are provided for a user who is familiar with the details of the method.

stat[5] = the entropy of $A^T * A$: : this is the Shannon entropy of $\text{diag}(A^T * A) / \text{Trace}(A^T * A)$ taken as point in the probability simplex.

stat[6] = the entropy of $A * A^T$.

istat

On exit,

istat[0] = the numerical rank determined after the initial QR factorization with pivoting. See the descriptions of *joba* and *jobr*.

istat[1] = the number of the computed nonzero singular value.

istat[2] = if nonzero, a warning message. If *istat*[2]=1, some of the column norms of *A* were denormalized floats. The requested high accuracy is not warranted by the data.

For complex flavors, *istat*[3] = 1 or -1. If *istat*[3] = 1, then the procedure used A^H to do the job as specified by the *job* parameters.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, the function did not converge in the maximal number of sweeps. The computed values may be inaccurate.

See Also

[?geqp3](#)

[?geqrf](#)

[?gelqf](#)

[?gesvj](#)

[?lamch](#)

[?pocon](#)

[?ormlq](#)

[?gesvj](#)

Computes the singular value decomposition of a real matrix using Jacobi plane rotations.

Syntax

```
lapack_int LAPACKESgesvj (int matrix_layout, char joba, char jobu, char jobv,
lapack_int m, lapack_int n, float * a, lapack_int lda, float * sva, lapack_int mv, float
* v, lapack_int ldv, float * stat);
```

```
lapack_int LAPACKEdgesvj (int matrix_layout, char joba, char jobu, char jobv,
lapack_int m, lapack_int n, double * a, lapack_int lda, double * sva, lapack_int mv,
double * v, lapack_int ldv, double * stat);
```

```
lapack_int LAPACKEcgesvj (int matrix_layout, char joba, char jobu, char jobv,
lapack_int m, lapack_int n, lapack_complex_float * a, lapack_int lda, float * sva,
lapack_int mv, lapack_complex_float * v, lapack_int ldv, float * stat);
```

```
lapack_int LAPACKE_zgesvj (int matrix_layout, char joba, char jobu, char jobv,
lapack_int m, lapack_int n, lapack_complex_double * a, lapack_int lda, double * sva,
lapack_int mv, lapack_complex_double * v, lapack_int ldv, double * stat);
```

Include Files

- mkl.h

Description

The routine computes the singular value decomposition (SVD) of a real or complex m -by- n matrix A , where $m \geq n$.

The SVD of A is written as

$A = U \Sigma^* V^T$ for real flavors, or

$A = U \Sigma^* V^H$ for complex flavors,

where Σ is an m -by- n diagonal matrix, U is an m -by- n orthonormal matrix, and V is an n -by- n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A ; the columns of U and V are the left and right singular vectors of A , respectively. The matrices U and V are computed and stored in the arrays u and v , respectively. The diagonal of Σ is computed and stored in the array sva .

The `?gesvj` routine can sometimes compute tiny singular values and their singular vectors much more accurately than other SVD routines.

The n -by- n orthogonal matrix V is obtained as a product of Jacobi plane rotations. The rotations are implemented as fast scaled rotations of Anda and Park [AndaPark94]. In the case of underflow of the Jacobi angle, a modified Jacobi transformation of Drmac ([Drmac08-4]) is used. Pivot strategy uses column interchanges of de Rijk ([deRijk98]). The relative accuracy of the computed singular values and the accuracy of the computed singular vectors (in angle metric) is as guaranteed by the theory of Demmel and Veselic [Demmel92]. The condition number that determines the accuracy in the full rank case is essentially

$$\left(\min_i d_{ii} \right) \cdot \kappa(A \cdot D)$$

where $\kappa(\cdot)$ is the spectral condition number. The best performance of this Jacobi SVD procedure is achieved if used in an accelerated version of Drmac and Veselic [Drmac08-1], [Drmac08-2].

The computational range for the nonzero singular values is the machine number interval $(\text{UNDERFLOW}, \text{OVERFLOW})$. In extreme cases, even denormalized singular values can be computed with the corresponding gradual loss of accurate digit.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>joba</i>	Must be 'L', 'U' or 'G'. Specifies the structure of A : If <i>joba</i> = 'L', the input matrix A is lower triangular.

	<p>If <code>joba = 'U'</code>, the input matrix <i>A</i> is upper triangular.</p> <p>If <code>joba = 'G'</code>, the input matrix <i>A</i> is a general <i>m</i>-by-<i>n</i>, $m \geq n$.</p> <p>Must be 'U', 'C' or 'N'.</p> <p>Specifies whether to compute the left singular vectors (columns of <i>U</i>):</p> <p>If <code>jobu = 'U'</code>, the left singular vectors corresponding to the nonzero singular values are computed and returned in the leading columns of <i>A</i>. See more details in the description of <i>a</i>. The default numerical orthogonality threshold is set to approximately $TOL = CTOL * EPS$, $CTOL = \sqrt{m}$, $EPS = ?lamch('E')$</p> <p>If <code>jobu = 'C'</code>, analogous to <code>jobu = 'U'</code>, except that you can control the level of numerical orthogonality of the computed left singular vectors. <i>TOL</i> can be set to $TOL = CTOL * EPS$, where <i>CTOL</i> is given on input in the array <i>stat</i>. No <i>CTOL</i> smaller than ONE is allowed. <i>CTOL</i> greater than $1 / EPS$ is meaningless. The option 'C' can be used if $m * EPS$ is satisfactory orthogonality of the computed left singular vectors, so $CTOL = m$ could save a few sweeps of Jacobi rotations. See the descriptions of <i>a</i> and <i>stat</i>[0].</p> <p>If <code>jobu = 'N'</code>, <i>u</i> is not computed. However, see the description of <i>a</i>.</p>
<i>jobv</i>	<p>Must be 'V', 'A' or 'N'.</p> <p>Specifies whether to compute the right singular vectors, that is, the matrix <i>V</i>:</p> <p>If <code>jobv = 'V'</code>, the matrix <i>V</i> is computed and returned in the array <i>v</i>.</p> <p>If <code>jobv = 'A'</code>, the Jacobi rotations are applied to the <i>mv</i>-by-<i>n</i> array <i>v</i>. In other words, the right singular vector matrix <i>V</i> is not computed explicitly, instead it is applied to an <i>mv</i>-by-<i>n</i> matrix initially stored in the first <i>mv</i> rows of <i>V</i>.</p> <p>If <code>jobv = 'N'</code>, the matrix <i>V</i> is not computed and the array <i>v</i> is not referenced.</p>
<i>m</i>	<p>The number of rows of the input matrix <i>A</i>.</p> <p>$1 / \text{slamch}('E') > m \geq 0$ for <i>sge</i>svj.</p> <p>$1 / \text{dlamch}('E') > m \geq 0$ for <i>dge</i>svj.</p>
<i>n</i>	<p>The number of columns in the input matrix <i>A</i>; $m \geq n \geq 0$.</p>
<i>a</i> , <i>v</i>	<p>Array <i>a</i> (size at least $lda * n$ for column major layout and $lda * m$ for row major layout) is an array containing the <i>m</i>-by-<i>n</i> matrix <i>A</i>.</p> <p>Array <i>v</i> (size at least $\max(1, ldv * n)$) contains, if <code>jobv = 'A'</code> the <i>mv</i>-by-<i>n</i> matrix to be post-multiplied by Jacobi rotations.</p>
<i>lda</i>	<p>The leading dimension of the array <i>a</i>. Must be at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.</p>
<i>mv</i>	<p>If <code>jobv = 'A'</code>, the product of Jacobi rotations in <i>?gesvj</i> is applied to the first <i>mv</i> rows of <i>v</i>. See the description of <i>jobv</i>. $0 \leq mv \leq ldv$.</p>
<i>ldv</i>	<p>The leading dimension of the array <i>v</i>; $ldv \geq 1$.</p> <p><code>jobv = 'V'</code>, $ldv \geq \max(1, n)$.</p>

$jobv = 'A'$, $ldv \geq \max(1, mv)$ for column major layout and $ldv \geq \max(1, n)$ for row major layout.

stat

Array size 6. If $jobu = 'C'$, $stat[0] = CTOL$, where $CTOL$ defines the threshold for convergence. The process stops if all columns of A are mutually orthogonal up to $CTOL * EPS$, where $EPS = ?lamch('E')$. It is required that $CTOL \geq 1$ - that is, it is not allowed to force the routine to obtain orthogonality below ϵ .

Output Parameters

a

On exit:

If $jobu = 'U'$ or $jobu = 'C'$:

- if $info = 0$, the leading columns of A contain left singular vectors corresponding to the computed singular values of a that are above the underflow threshold $?lamch('S')$, that is, non-zero singular values. The number of the computed non-zero singular values is returned in $stat[1]$. Also see the descriptions of *sva* and *stat*. The computed columns of u are mutually numerically orthogonal up to approximately $TOL = \sqrt{m} * EPS$ (default); or $TOL = CTOL * EPS$ $jobu = 'C'$, see the description of *jobu*.
- if $info > 0$, the procedure `?gesvj` did not converge in the given number of iterations (sweeps). In that case, the computed columns of u may not be orthogonal up to TOL . The output u (stored in *a*), *sigma* (given by the computed singular values in $sva(1:n)$) and v is still a decomposition of the input matrix A in the sense that the residual $\|A - scale * U * sigma * V^T\|_2 / \|A\|_2$ for real flavors or $\|A - scale * U * sigma * V^H\|_2 / \|A\|_2$ for complex flavors (where $scale = stat[0]$) is small.

If $jobu = 'N'$:

- if $info = 0$, note that the left singular vectors are 'for free' in the one-sided Jacobi SVD algorithm. However, if only the singular values are needed, the level of numerical orthogonality of u is not an issue and iterations are stopped when the columns of the iterated matrix are numerically orthogonal up to approximately $m * EPS$. Thus, on exit, *a* contains the columns of u scaled with the corresponding singular values.
- if $info > 0$, the procedure `?gesvj` did not converge in the given number of iterations (sweeps).

sva

Array size n .

If $info = 0$, depending on the value $scale = stat[0]$, where $scale$ is the scaling factor:

- if $scale = 1$, $sva[0:n - 1]$ contains the computed singular values of a .
- if $scale \neq 1$, the singular values of a are $scale * sva(1:n)$, and this factored representation is due to the fact that some of the singular values of a might underflow or overflow.

If $info > 0$, the procedure `?gesvj` did not converge in the given number of iterations (sweeps) and $scale * sva(1:n)$ may not be accurate.

<code>v</code>	<p>On exit:</p> <p>If <code>jobv = 'V'</code>, contains the n-by-n matrix of the right singular vectors.</p> <p>If <code>jobv = 'A'</code>, then <code>v</code> contains the product of the computed right singular vector matrix and the initial matrix in the array <code>v</code>.</p> <p>If <code>jobv = 'N'</code>, <code>v</code> is not referenced.</p>
<code>stat</code>	<p>On exit,</p> <p><code>stat[0] = scale</code> is the scaling factor such that <code>scale*sva(1:n)</code> are the computed singular values of <code>A</code>. See the description of <code>sva</code>.</p> <p><code>stat[1]</code> is the number of the computed nonzero singular values.</p> <p><code>stat[2]</code> is the number of the computed singular values that are larger than the underflow threshold.</p> <p><code>stat[3]</code> is the number of sweeps of Jacobi rotations needed for numerical convergence.</p> <p><code>stat[4] = max_{i≠j} COS(A(:,i),A(:,j)) </code> in the last sweep. This is useful information in cases when <code>?gesvj</code> did not converge, as it can be used to estimate whether the output is still useful and for post festum analysis.</p> <p><code>stat[5]</code> is the largest absolute value over all sines of the Jacobi rotation angles in the last sweep. It can be useful in a post festum analysis.</p>

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info > 0`, the function did not converge in the maximal number (30) of sweeps. The output may still be useful. See the description of `stat`.

`?ggsvd`

Computes the generalized singular value decomposition of a pair of general rectangular matrices (deprecated).

Syntax

```
lapack_int LAPACKE_sggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l, float* a,
lapack_int lda, float* b, lapack_int ldb, float* alpha, float* beta, float* u,
lapack_int ldu, float* v, lapack_int ldv, float* q, lapack_int ldq, lapack_int* iwork );

lapack_int LAPACKE_dggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l, double* a,
lapack_int lda, double* b, lapack_int ldb, double* alpha, double* beta, double* u,
lapack_int ldu, double* v, lapack_int ldv, double* q, lapack_int ldq, lapack_int*
iwork );

lapack_int LAPACKE_cggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
```


$$D_2 = \begin{matrix} & k & 1 \\ & 1 & \\ p-1 & \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{matrix}$$

$$\begin{matrix} & n-k-1 & k & 1 \\ (0 \ R) = & \begin{matrix} k \\ 1 \end{matrix} \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix}, \end{matrix}$$

where

$C = \text{diag}(\text{alpha}[k], \dots, \text{alpha}[k + 1 - 1])$

$S = \text{diag}(\text{beta}[k], \dots, \text{beta}[k + 1 - 1])$

$C^2 + S^2 = I$

Nonzero element r_{ij} ($1 \leq j \leq k + 1$) of R is stored in $a[(i - 1) + (n - k - 1 + j - 1) * lda]$ for column major layout and in $a[(i - 1) * lda + (n - k - 1 + j - 1)]$ for row major layout.

If $m - k - 1 < 0$,

$$D_1 = \begin{matrix} & k & m-k & k+1-m \\ & & & \\ m-k & \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{matrix}$$

$$D_2 = \begin{matrix} & k & m-k & k+1-m \\ \begin{matrix} m-k \\ k+1-m \\ p-1 \end{matrix} & \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{matrix}$$

$$(0 \ R) = \begin{matrix} & n-k-1 & k & m-k & k+1-m \\ \begin{matrix} k \\ m-k \\ k+1-m \end{matrix} & \begin{pmatrix} 0 & R_{11} & R_{12} & R_{13} \\ 0 & 0 & R_{22} & R_{23} \\ 0 & 0 & 0 & R_{33} \end{pmatrix} \end{matrix}$$

where

$C = \text{diag}(\alpha[k], \dots, \alpha[m]),$
 $S = \text{diag}(\beta[k], \dots, \beta[m-1]),$
 $C_2 + S_2 = I$

On exit, the location of nonzero element r_{ij} ($1 \leq j \leq k+1$) of R depends on the value of i . For $i \leq m$ this element is stored in $a[(i-1) + (n-k-1+j-1)*lda]$ for column major layout and in $a[(i-1)*lda + (n-k-1+j-1)]$ for row major layout. For $m < i \leq k+1$ it is stored in $b[(i-k-1) + (n-k-1+j-1)*ldb]$ for column major layout and in $b[(i-k-1)*ldb + (n-k-1+j-1)]$ for row major layout.

The routine computes C , S , R , and optionally the orthogonal/unitary transformation matrices U , V and Q .

In particular, if B is an n -by- n nonsingular matrix, then the GSVD of A and B implicitly gives the SVD of $A*B^{-1}$:

$$A*B^{-1} = U*(D_1*D_2^{-1})*V'.$$

If $(A', B)'$ has orthonormal columns, then the GSVD of A and B is also equal to the CS decomposition of A and B . Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

$$A'*A*x = \lambda*B'*B*x.$$

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobu</i>	Must be 'U' or 'N'. If <i>jobu</i> = 'U', orthogonal/unitary matrix U is computed. If <i>jobu</i> = 'N', U is not computed.
<i>jobv</i>	Must be 'V' or 'N'. If <i>jobv</i> = 'V', orthogonal/unitary matrix V is computed.

	If $jobv = 'N'$, V is not computed.
$jobq$	Must be $'Q'$ or $'N'$. If $jobq = 'Q'$, orthogonal/unitary matrix Q is computed. If $jobq = 'N'$, Q is not computed.
m	The number of rows of the matrix A ($m \geq 0$).
n	The number of columns of the matrices A and B ($n \geq 0$).
p	The number of rows of the matrix B ($p \geq 0$).
a, b	Arrays: a (size at least $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout) contains the m -by- n matrix A . b (size at least $\max(1, ldb*n)$ for column major layout and $\max(1, ldb*p)$ for row major layout) contains the p -by- n matrix B .
lda	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
ldb	The leading dimension of b ; at least $\max(1, p)$ for column major layout and $\max(1, n)$ for row major layout.
ldu	The leading dimension of the array u . $ldu \geq \max(1, m)$ if $jobu = 'U'$; $ldu \geq 1$ otherwise.
ldv	The leading dimension of the array v . $ldv \geq \max(1, p)$ if $jobv = 'V'$; $ldv \geq 1$ otherwise.
ldq	The leading dimension of the array q . $ldq \geq \max(1, n)$ if $jobq = 'Q'$; $ldq \geq 1$ otherwise.

Output Parameters

k, l	On exit, k and l specify the dimension of the subblocks. The sum $k+l$ is equal to the effective numerical rank of $(A', B)'$.
a	On exit, a contains the triangular matrix R or part of R .
b	On exit, b contains part of the triangular matrix R if $m-k-l < 0$.
$alpha, beta$	Arrays, size at least $\max(1, n)$ each. Contain the generalized singular value pairs of A and B : $alpha(1:k) = 1,$ $beta(1:k) = 0,$ and if $m-k-l \geq 0,$ $alpha(k+1:k+l) = C,$ $beta(k+1:k+l) = S,$ or if $m-k-l < 0,$

```

alpha(k+1:m) = C, alpha(m+1:k+1) = 0
beta(k+1:m) = S, beta(m+1:k+1) = 1
and
alpha(k+1+1:n) = 0
beta(k+1+1:n) = 0.

```

u, v, q

Arrays:

u, size at least $\max(1, ldu * m)$.

If *jobu* = 'U', *u* contains the *m*-by-*m* orthogonal/unitary matrix *U*.

If *jobu* = 'N', *u* is not referenced.

v, size at least $\max(1, ldv * p)$.

If *jobv* = 'V', *v* contains the *p*-by-*p* orthogonal/unitary matrix *V*.

If *jobv* = 'N', *v* is not referenced.

q, size at least $\max(1, ldq * n)$.

If *jobq* = 'Q', *q* contains the *n*-by-*n* orthogonal/unitary matrix *Q*.

If *jobq* = 'N', *q* is not referenced.

iwork

On exit, *iwork* stores the sorting information.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, the Jacobi-type procedure failed to converge. For further details, see subroutine [tgsja](#).

?gesvdx

Computes the SVD and left and right singular vectors for a matrix.

Syntax

```

lapack_int LAPACKESgesvdx (int matrix_layout, char jobu, char jobvt, char range,
lapack_int m, lapack_int n, float * a, lapack_int lda, float vl, float vu, lapack_int
il, lapack_int iu, lapack_int * ns, float * s, float * u, lapack_int ldu, float * vt,
lapack_int ldvt, lapack_int * superb);

```

```

lapack_int LAPACKEdgesvdx (int matrix_layout, char jobu, char jobvt, char range,
lapack_int m, lapack_int n, double * a, lapack_int lda, double vl, double vu, lapack_int
il, lapack_int iu, lapack_int * ns, double * s, double * u, lapack_int ldu, double * vt,
lapack_int ldvt, lapack_int * superb);

```

```

lapack_int LAPACKEcgesvdx (int matrix_layout, char jobu, char jobvt, char range,
lapack_int m, lapack_int n, lapack_complex_float * a, lapack_int lda, float vl, float
vu, lapack_int il, lapack_int iu, lapack_int * ns, float * s, lapack_complex_float * u,
lapack_int ldu, lapack_complex_float * vt, lapack_int ldvt, lapack_int * superb);

```

```
lapack_int LAPACKE_zgesvdx (int matrix_layout, char jobu, char jobvt, char range,
lapack_int m, lapack_int n, lapack_complex_double * a, lapack_int lda, double vl,
double vu, lapack_int il, lapack_int iu, lapack_int * ns, double * s,
lapack_complex_double * u, lapack_int ldu, lapack_complex_double * vt, lapack_int ldvt,
lapack_int * superb);
```

Include Files

- mkl.h

Description

`?gesvdx` computes the singular value decomposition (SVD) of a real or complex m -by- n matrix A , optionally computing the left and right singular vectors. The SVD is written

$$A = U * \Sigma * \text{transpose}(V)$$

where Σ is an m -by- n matrix which is zero except for its $\min(m,n)$ diagonal elements, U is an m -by- m matrix, and V is an n -by- n matrix. The matrices U and V are orthogonal for real A , and unitary for complex A . The diagonal elements of Σ are the singular values of A ; they are real and non-negative, and are returned in descending order. The first $\min(m,n)$ columns of U and V are the left and right singular vectors of A .

`?gesvdx` uses an eigenvalue problem for obtaining the SVD, which allows for the computation of a subset of singular values and vectors. See `?bdsvdx` for details.

Note that the routine returns V^T , not V .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobu</i>	Specifies options for computing all or part of the matrix U : = 'V': the first $\min(m,n)$ columns of U (the left singular vectors) or as specified by <i>range</i> are returned in the array <i>u</i> ; = 'N': no columns of U (no left singular vectors) are computed.
<i>jobvt</i>	Specifies options for computing all or part of the matrix V^T : = 'V': the first $\min(m,n)$ rows of V^T (the right singular vectors) or as specified by <i>range</i> are returned in the array <i>vt</i> ; = 'N': no rows of V^T (no right singular vectors) are computed.
<i>range</i>	= 'A': find all singular values. = 'V': all singular values in the half-open interval $(vl, vu]$ are found. = 'I': the <i>il</i> -th through <i>iu</i> -th singular values are found.
<i>m</i>	The number of rows of the input matrix A . $m \geq 0$.
<i>n</i>	The number of columns of the input matrix A . $n \geq 0$.
<i>a</i>	Array, size $lda*n$ On entry, the m -by- n matrix A .
<i>lda</i>	The leading dimension of the array <i>a</i> . $lda \geq \max(1, m)$.

<i>vl</i>	$vl \geq 0$.
<i>vu</i>	If <i>range</i> ='V', the lower and upper bounds of the interval to be searched for singular values. $vu > vl$. Not referenced if <i>range</i> = 'A' or 'I'.
<i>il</i>	
<i>iu</i>	If <i>range</i> ='I', the indices (in ascending order) of the smallest and largest singular values to be returned. $1 \leq il \leq iu \leq \min(m, n)$, if $\min(m, n) > 0$. Not referenced if <i>range</i> = 'A' or 'V'.
<i>ldu</i>	The leading dimension of the array <i>u</i> . $ldu \geq 1$; if <i>jobu</i> = 'V', $ldu \geq m$.
<i>ldvt</i>	The leading dimension of the array <i>vt</i> . $ldvt \geq 1$; if <i>jobvt</i> = 'V', $ldvt \geq ns$ (see above).

Output Parameters

<i>a</i>	On exit, the contents of <i>a</i> are destroyed.
<i>ns</i>	The total number of singular values found, $0 \leq ns \leq \min(m, n)$. If <i>range</i> = 'A', $ns = \min(m, n)$; if <i>range</i> = 'I', $ns = iu - il + 1$.
<i>s</i>	Array, size $\min(m, n)$ The singular values of <i>A</i> , sorted so that $s[i] \geq s[i + 1]$.
<i>u</i>	Array, size $ldu * ucol$ If <i>jobu</i> = 'V', <i>u</i> contains columns of <i>U</i> (the left singular vectors, stored columnwise) as specified by <i>range</i> ; if <i>jobu</i> = 'N', <i>u</i> is not referenced.

NOTE

Make sure that $ucol \geq ns$; if *range* = 'V', the exact value of *ns* is not known in advance and an upper bound must be used.

<i>vt</i>	Array, size $ldvt * n$ If <i>jobvt</i> = 'V', <i>vt</i> contains the rows of V^T (the right singular vectors, stored rowwise) as specified by <i>range</i> ; if <i>jobvt</i> = 'N', <i>vt</i> is not referenced.
-----------	---

NOTE

Make sure that $ldvt \geq ns$; if *range* = 'V', the exact value of *ns* is not known in advance and an upper bound must be used.

<i>superb</i>	Array, size $(12 * \min(m, n))$. If <i>info</i> = 0, the first <i>ns</i> elements of <i>superb</i> are zero. If <i>info</i> > 0, then <i>superb</i> contains the indices of the eigenvectors that failed to converge in ?bdsvdX/?stevX.
---------------	---

Return Values

This function returns a value *info*.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value.

> 0: if *info* = *i*, then *i* eigenvectors failed to converge in ?bdsvd/?stevx. if *info* = *n**2 + 1, an internal error occurred in ?bdsvd.

?bdsvd

Computes the SVD of a bidiagonal matrix.

Syntax

```
lapack_int LAPACKE_sbdsvd (int matrix_layout, char uplo, char jobz, char range,
lapack_int n, float * d, float * e, float vl, float vu, lapack_int il, lapack_int iu,
lapack_int * ns, float * s, float * z, lapack_int ldz, lapack_int * superb);
```

```
lapack_int LAPACKE_dbdsvd (int matrix_layout, char uplo, char jobz, char range,
lapack_int n, double * d, double * e, double vl, double vu, lapack_int il, lapack_int
iu, lapack_int * ns, double * s, double * z, lapack_int ldz, lapack_int * superb);
```

Include Files

- mkl.h

Description

?bdsvd computes the singular value decomposition (SVD) of a real *n*-by-*n* (upper or lower) bidiagonal matrix *B*, $B = U * S * VT$, where *S* is a diagonal matrix with non-negative diagonal elements (the singular values of *B*), and *U* and *VT* are orthogonal matrices of left and right singular vectors, respectively.

Given an upper bidiagonal *B* with diagonal $d = [d_1 d_2 \dots d_n]$ and superdiagonal $e = [e_1 e_2 \dots e_{n-1}]$, *?bdsvd* computes the singular value decomposition of *B* through the eigenvalues and eigenvectors of the *n**2-by-*n**2 tridiagonal matrix

$$TGK = \begin{pmatrix} 0 & d_1 & & & \\ d_1 & 0 & e_1 & & \\ & e_1 & 0 & d_2 & \\ & & d_2 & \ddots & \ddots \\ & & & \ddots & \ddots \end{pmatrix}$$

If (s, u, v) is a singular triplet of *B* with $\|u\| = \|v\| = 1$, then $(\pm s, q)$, $\|q\| = 1$, are eigenpairs of TGK, with

$$q = P * \frac{(u' \pm v')}{\sqrt{2}} = \frac{(v_1 \ u_1 \ v_2 \ u_2 \ \dots \ v_n \ u_n)}{\sqrt{2}}, \text{ and } P = (e_{n+1} \ e_1 \ e_{n+2} \ e_2 \ \dots).$$

Given a TGK matrix, one can either

1. compute -*s*, -*v* and change signs so that the singular values (and corresponding vectors) are already in descending order (as in ?gesvd/?gesdd) or
2. compute *s*, *v* and reorder the values (and corresponding vectors).

?bdsvd implements (1) by calling ?stevx (bisection plus inverse iteration, to be replaced with a version of the Multiple Relative Robust Representation algorithm. (See P. Willems and B. Lang, A framework for the MR³ algorithm: theory and implementation, SIAM J. Sci. Comput., 35:740-766, 2013.)

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>uplo</i>	= 'U': <i>B</i> is upper bidiagonal; = 'L': <i>B</i> is lower bidiagonal.
<i>jobz</i>	= 'N': Compute singular values only; = 'V': Compute singular values and singular vectors.
<i>range</i>	= 'A': Find all singular values. = 'V': all singular values in the half-open interval $[v_l, v_u)$ are found. = 'I': the <i>il</i> -th through <i>iu</i> -th singular values are found.
<i>n</i>	The order of the bidiagonal matrix. $n \geq 0$.
<i>d</i>	Array, size <i>n</i> . The <i>n</i> diagonal elements of the bidiagonal matrix <i>B</i> .
<i>e</i>	Array, size $\max(1, n - 1)$ The $(n - 1)$ superdiagonal elements of the bidiagonal matrix <i>B</i> in elements 1 to <i>n</i> - 1.
<i>vl</i>	$vl \geq 0$.
<i>vu</i>	If <i>range</i> ='V', the lower and upper bounds of the interval to be searched for singular values. $vu > vl$. Not referenced if <i>range</i> = 'A' or 'I'.
<i>il, iu</i>	If <i>range</i> ='I', the indices (in ascending order) of the smallest and largest singular values to be returned. $1 \leq il \leq iu \leq \min(m, n)$, if $\min(m, n) > 0$. Not referenced if <i>range</i> = 'A' or 'V'.
<i>ldz</i>	The leading dimension of the array <i>z</i> . $ldz \geq 1$, and if <i>jobz</i> = 'V', $ldz \geq \max(2, n*2)$.

Output Parameters

<i>ns</i>	The total number of singular values found. $0 \leq ns \leq n$. If <i>range</i> = 'A', <i>ns</i> = <i>n</i> , and if <i>range</i> = 'I', <i>ns</i> = <i>iu</i> - <i>il</i> + 1.
<i>s</i>	Array, size (<i>n</i>) The first <i>ns</i> elements contain the selected singular values in ascending order.
<i>z</i>	Array, size $2*n*k$

If `jobz = 'V'`, then if `info = 0` the first `ns` columns of `z` contain the singular vectors of the matrix `B` corresponding to the selected singular values, with `U` in rows 1 to `n` and `V` in rows `n+1` to `n*2`, i.e.

$$z = \begin{pmatrix} U \\ V \end{pmatrix}$$

If `jobz = 'N'`, then `z` is not referenced.

NOTE

Make sure that at least $k = ns + 1$ columns are supplied in the array `z`; if `range = 'V'`, the exact value of `ns` is not known in advance and an upper bound must be used.

`superb`

Array, size $(12 * n)$.

If `jobz = 'V'`, then if `info = 0`, the first `ns` elements of `iwork` are zero. If `info > 0`, then `iwork` contains the indices of the eigenvectors that failed to converge in `?stevx`.

Return Values

This function returns a value `info`.

= 0: successful exit.

< 0: if `info = -i`, the `i`-th argument had an illegal value.

> 0:

if `info = i`, then `i` eigenvectors failed to converge in `?stevx`. The indices of the eigenvectors (as returned by `?stevx`) are stored in the array `iwork`.

if `info = n*2 + 1`, an internal error occurred.

Cosine-Sine Decomposition: LAPACK Driver Routines

This topic describes LAPACK driver routines for computing the *cosine-sine decomposition* (CS decomposition). You can also call the corresponding computational routines to perform the same task.

The computation has the following phases:

1. The matrix is reduced to a bidiagonal block form.
2. The blocks are simultaneously diagonalized using techniques from the bidiagonal SVD algorithms.

Table "Driver Routines for Cosine-Sine Decomposition (CSD)" lists LAPACK routines that perform CS decomposition of matrices.

Computational Routines for Cosine-Sine Decomposition (CSD)

Operation	Real matrices	Complex matrices
Compute the CS decomposition of a block-partitioned orthogonal matrix	orcsd uncsd	
Compute the CS decomposition of a block-partitioned unitary matrix		orcsd uncsd

See Also

[CS Computational Routines](#)

?orcsd/?uncsd

Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

Syntax

```
lapack_int LAPACKE_sorcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, float* x11,
lapack_int ldx11, float* x12, lapack_int ldx12, float* x21, lapack_int ldx21, float*
x22, lapack_int ldx22, float* theta, float* u1, lapack_int ldu1, float* u2, lapack_int
ldu2, float* v1t, lapack_int ldv1t, float* v2t, lapack_int ldv2t );
```

```
lapack_int LAPACKE_dorcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, double* x11,
lapack_int ldx11, double* x12, lapack_int ldx12, double* x21, lapack_int ldx21, double*
x22, lapack_int ldx22, double* theta, double* u1, lapack_int ldu1, double* u2,
lapack_int ldu2, double* v1t, lapack_int ldv1t, double* v2t, lapack_int ldv2t );
```

```
lapack_int LAPACKE_cuncsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q,
lapack_complex_float* x11, lapack_int ldx11, lapack_complex_float* x12, lapack_int
ldx12, lapack_complex_float* x21, lapack_int ldx21, lapack_complex_float* x22,
lapack_int ldx22, float* theta, lapack_complex_float* u1, lapack_int ldu1,
lapack_complex_float* u2, lapack_int ldu2, lapack_complex_float* v1t, lapack_int ldv1t,
lapack_complex_float* v2t, lapack_int ldv2t );
```

```
lapack_int LAPACKE_zuncsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q,
lapack_complex_double* x11, lapack_int ldx11, lapack_complex_double* x12, lapack_int
ldx12, lapack_complex_double* x21, lapack_int ldx21, lapack_complex_double* x22,
lapack_int ldx22, double* theta, lapack_complex_double* u1, lapack_int ldu1,
lapack_complex_double* u2, lapack_int ldu2, lapack_complex_double* v1t, lapack_int
ldv1t, lapack_complex_double* v2t, lapack_int ldv2t );
```

Include Files

- mkl.h

Description

The routines ?orcsd/?uncsd compute the CS decomposition of an m -by- m partitioned orthogonal matrix X :

$$X = \left(\begin{array}{c|c} x_{11} & x_{12} \\ \hline x_{21} & x_{22} \end{array} \right) = \left(\begin{array}{c|c} u_1 & \\ \hline & u_2 \end{array} \right) \left(\begin{array}{ccc|ccc} I & 0 & 0 & 0 & 0 & 0 \\ 0 & C & 0 & 0 & -S & 0 \\ 0 & 0 & 0 & 0 & 0 & -I \\ \hline 0 & 0 & 0 & I & 0 & 0 \\ 0 & S & 0 & 0 & C & 0 \\ 0 & 0 & I & 0 & 0 & 0 \end{array} \right) \left(\begin{array}{c|c} v_1 & \\ \hline & v_2 \end{array} \right)^T$$

or unitary matrix:

$$X = \left(\begin{array}{c|c} x_{11} & x_{12} \\ \hline x_{21} & x_{22} \end{array} \right) = \left(\begin{array}{c|c} u_1 & \\ \hline & u_2 \end{array} \right) \left(\begin{array}{ccc|ccc} I & 0 & 0 & 0 & 0 & 0 \\ 0 & C & 0 & 0 & -S & 0 \\ 0 & 0 & 0 & 0 & 0 & -I \\ \hline 0 & 0 & 0 & I & 0 & 0 \\ 0 & S & 0 & 0 & C & 0 \\ 0 & 0 & I & 0 & 0 & 0 \end{array} \right) \left(\begin{array}{c|c} v_1 & \\ \hline & v_2 \end{array} \right)^H$$

x_{11} is p -by- q . The orthogonal/unitary matrices u_1 , u_2 , v_1 , and v_2 are p -by- p , $(m-p)$ -by- $(m-p)$, q -by- q , $(m-q)$ -by- $(m-q)$, respectively. C and S are r -by- r nonnegative diagonal matrices satisfying $C^2 + S^2 = I$, in which $r = \min(p, m-p, q, m-q)$.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobu1</code>	If equals <code>Y</code> , then u_1 is computed. Otherwise, u_1 is not computed.
<code>jobu2</code>	If equals <code>Y</code> , then u_2 is computed. Otherwise, u_2 is not computed.
<code>jobv1t</code>	If equals <code>Y</code> , then v_1^t is computed. Otherwise, v_1^t is not computed.
<code>jobv2t</code>	If equals <code>Y</code> , then v_2^t is computed. Otherwise, v_2^t is not computed.
<code>trans</code>	<div> <div>= 'T':</div> <div>x, u_1, u_2, v_1^t, v_2^t are stored in row-major order.</div> </div> <div> <div>otherwise</div> <div>x, u_1, u_2, v_1^t, v_2^t are stored in column-major order.</div> </div>
<code>signs</code>	<div> <div>= 'O':</div> <div>The lower-left block is made nonpositive (the "other" convention).</div> </div> <div> <div>otherwise</div> <div>The upper-right block is made nonpositive (the "default" convention).</div> </div>
<code>m</code>	The number of rows and columns of the matrix X .
<code>p</code>	The number of rows in x_{11} and x_{12} . $0 \leq p \leq m$.
<code>q</code>	The number of columns in x_{11} and x_{21} . $0 \leq q \leq m$.
<code>x11, x12, x21, x22</code>	<p>Arrays of size $x11$ ($ldx11, q$), $x12$ ($ldx12, m - q$), $x21$ ($ldx21, q$), and $x22$ ($ldx22, m - q$).</p> <p>Contain the parts of the orthogonal/unitary matrix whose CSD is desired.</p>
<code>ldx11, ldx12, ldx21, ldx22</code>	The leading dimensions of the parts of array X . $ldx11 \geq \max(1, p)$, $ldx12 \geq \max(1, p)$, $ldx21 \geq \max(1, m - p)$, $ldx22 \geq \max(1, m - p)$.
<code>ldu1</code>	The leading dimension of the array u_1 . If <code>jobu1</code> = 'Y', $ldu1 \geq \max(1, p)$.
<code>ldu2</code>	The leading dimension of the array u_2 . If <code>jobu2</code> = 'Y', $ldu2 \geq \max(1, m - p)$.
<code>ldv1t</code>	The leading dimension of the array $v1t$. If <code>jobv1t</code> = 'Y', $ldv1t \geq \max(1, q)$.

ldv2t The leading dimension of the array *v2t*. If *jobv2t* = 'Y', $ldv2t \geq \max(1, m-q)$.

Output Parameters

theta Array, size *r*, in which $r = \min(p, m-p, q, m-q)$.
 $C = \text{diag}(\cos(\theta[0]), \dots, \cos(\theta[r-1]))$, and
 $S = \text{diag}(\sin(\theta[0]), \dots, \sin(\theta[r-1]))$.

u1 Array, size at least $\max(1, ldu1 * p)$.
 If *jobu1* = 'Y', *u1* contains the *p*-by-*p* orthogonal/unitary matrix u_1 .

u2 Array, size at least $\max(1, ldu2 * (m-p))$.
 If *jobu2* = 'Y', *u2* contains the $(m-p)$ -by- $(m-p)$ orthogonal/unitary matrix u_2 .

v1t Array, size at least $\max(1, ldv1t * q)$.
 If *jobv1t* = 'Y', *v1t* contains the *q*-by-*q* orthogonal matrix v_1^T or unitary matrix v_1^H .

v2t Array, size at least $\max(1, ldv2t * (m-q))$.
 If *jobv2t* = 'Y', *v2t* contains the $(m-q)$ -by- $(m-q)$ orthogonal matrix v_2^T or unitary matrix v_2^H .

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

> 0: ?orcsd/?uncsd did not converge.

See Also

[?bbcsd](#)

[xerbla](#)

[?orcsd2by1/?uncsd2by1](#)

Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

Syntax

```
lapack_int LAPACKE_sorcsd2by1 (int matrix_layout, char jobu1, char jobu2, char jobv1t,
lapack_int m, lapack_int p, lapack_int q, float * x11, lapack_int ldx11, float * x21,
lapack_int ldx21, float * theta, float * u1, lapack_int ldu1, float * u2, lapack_int
ldu2, float * v1t, lapack_int ldv1t);
```

```
lapack_int LAPACKE_dorcsd2by1 (int matrix_layout, char jobu1, char jobu2, char jobv1t,
lapack_int m, lapack_int p, lapack_int q, double * x11, lapack_int ldx11, double * x21,
lapack_int ldx21, double * theta, double * u1, lapack_int ldu1, double * u2, lapack_int
ldu2, double * v1t, lapack_int ldv1t);
```

```
lapack_int LAPACKE_cuncsd2by1 (int matrix_layout, char jobu1, char jobu2, char jobv1t,
lapack_int m, lapack_int p, lapack_int q, lapack_complex_float * x11, lapack_int ldx11,
lapack_complex_float * x21, lapack_int ldx21, float * theta, lapack_complex_float * u1,
lapack_int ldu1, lapack_complex_float * u2, lapack_int ldu2, lapack_complex_float *
v1t, lapack_int ldv1t);
```

```
lapack_int LAPACKE_zuncsd2by1 (int matrix_layout, char jobu1, char jobu2, char jobv1t,
lapack_int m, lapack_int p, lapack_int q, lapack_complex_double * x11, lapack_int
ldx11, lapack_complex_double * x21, lapack_int ldx21, double * theta,
lapack_complex_double * u1, lapack_int ldu1, lapack_complex_double * u2, lapack_int
ldu2, lapack_complex_double * v1t, lapack_int ldv1t);
```

Include Files

- mkl.h

Description

The routines ?orcsd2by1/?uncsd2by1 compute the CS decomposition of an m -by- q matrix X with orthonormal columns that has been partitioned into a 2-by-1 block structure:

$$X = \begin{bmatrix} X_{11} \\ X_{21} \end{bmatrix} = \begin{bmatrix} U_1 & | & \\ \hline & & U_2 \end{bmatrix} \begin{bmatrix} I & 0 & 0 \\ 0 & C & 0 \\ 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & S & 0 \\ 0 & 0 & I \end{bmatrix} V_1^H$$

X_{11} is p -by- q . The orthogonal/unitary matrices u_1 , u_2 , v_1 , and v_2 are p -by- p , $(m-p)$ -by- $(m-p)$, q -by- q , $(m-q)$ -by- $(m-q)$, respectively. C and S are r -by- r nonnegative diagonal matrices satisfying $C^2 + S^2 = I$, in which $r = \min(p, m-p, q, m-q)$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobu1</i>	If equal to 'Y', then u_1 is computed. Otherwise, u_1 is not computed.
<i>jobu2</i>	If equal to 'Y', then u_2 is computed. Otherwise, u_2 is not computed.
<i>jobv1t</i>	If equal to 'Y', then v_1^t is computed. Otherwise, v_1^t is not computed.
<i>m</i>	The number of rows and columns of the matrix X .

p	The number of rows in x_{11} . $0 \leq p \leq m$.
q	The number of columns in x_{11} . $0 \leq q \leq m$.
x_{11}	Array, size $(ldx11*q)$. On entry, the part of the orthogonal matrix whose CSD is desired.
$ldx11$	The leading dimension of the array x_{11} . $ldx11 \geq \max(1, p)$.
x_{21}	Array, size $(ldx21*q)$. On entry, the part of the orthogonal matrix whose CSD is desired.
$ldx21$	The leading dimension of the array X . $ldx21 \geq \max(1, m - p)$.
$ldu1$	The leading dimension of the array u_1 . If $jobu1 = 'Y'$, $ldu1 \geq \max(1, p)$.
$ldu2$	The leading dimension of the array u_2 . If $jobu2 = 'Y'$, $ldu2 \geq \max(1, m-p)$.
$ldv1t$	The leading dimension of the array $v1t$. If $jobv1t = 'Y'$, $ldv1t \geq \max(1, q)$.

Output Parameters

θ	Array, size r , in which $r = \min(p, m-p, q, m-q)$. $C = \text{diag}(\cos(\theta(1)), \dots, \cos(\theta(r)))$, and $S = \text{diag}(\sin(\theta(1)), \dots, \sin(\theta(r)))$.
u_1	Array, size $(ldu1*p)$. If $jobu1 = 'Y'$, u_1 contains the p -by- p orthogonal/unitary matrix u_1 .
u_2	Array, size $(ldu2*(m - p))$. If $jobu2 = 'Y'$, u_2 contains the $(m-p)$ -by- $(m-p)$ orthogonal/unitary matrix u_2 .
$v1t$	Array, size $(ldv1t*q)$. If $jobv1t = 'Y'$, $v1t$ contains the q -by- q orthogonal matrix v_1^T or unitary matrix v_1^H .

Return Values

This function returns a value *info*.

= 0: successful exit

< 0: if *info* = $-i$, the i -th argument has an illegal value

> 0: ?orcsd2by1/?uncsd2by1 did not converge.

See Also

[?bbcsd](#)

[xerbla](#)

Generalized Symmetric Definite Eigenvalue Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving generalized symmetric definite eigenproblems. See also [computational routines](#) that can be called to solve these problems. [Table "Driver Routines for Solving Generalized Symmetric Definite Eigenproblems"](#) lists all such driver routines.

Driver Routines for Solving Generalized Symmetric Definite Eigenproblems

Routine Name	Operation performed
sygv/hegv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem.
sygvd/hegvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.
sygvx/hegvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem.
spgv/hpgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage.
spgvd/hpgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.
spgvx/hpgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage.
sbgv/hbgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices.
sbgvd/hbgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.
sbgvx/hbgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices.

?sygv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
lapack_int LAPACKE_ssygv (int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float* w);
```

```
lapack_int LAPACKE_dsygv (int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double* w);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be symmetric and B is also positive definite.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>itype</i>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of A and B .
<i>n</i>	The order of the matrices A and B ($n \geq 0$).
<i>a, b</i>	Arrays: <i>a</i> (size at least $\max(1, lda*n)$) contains the upper or lower triangle of the symmetric matrix A , as specified by <i>uplo</i> . <i>b</i> (size at least $\max(1, ldb*n)$) contains the upper or lower triangle of the symmetric positive definite matrix B , as specified by <i>uplo</i> .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit, if <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>a</i> contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^T*B*Z = I$; if <i>itype</i> = 3, $Z^T*inv(B)*Z = I$; If <i>jobz</i> = 'N', then on exit the upper triangle (if <i>uplo</i> = 'U') or the lower triangle (if <i>uplo</i> = 'L') of A , including the diagonal, is destroyed.
<i>b</i>	On exit, if $info \leq n$, the part of <i>b</i> containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^T*U$ or $B = L*L^T$.
<i>w</i>	Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, *spotrf/dpotrf* or *ssyev/dsyev* returned an error code:

If *info* = *i* ≤ *n*, *ssyev/dsyev* failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

?hegv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

```
lapack_int LAPACKC_chegv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b,
lapack_int ldb, float* w );
```

```
lapack_int LAPACKC_zhegv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b,
lapack_int ldb, double* w );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A*x = \lambda*B*x, \quad A*B*x = \lambda*x, \quad \text{or} \quad B*A*x = \lambda*x.$$

Here *A* and *B* are assumed to be Hermitian and *B* is also positive definite.

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

itype Must be 1 or 2 or 3. Specifies the problem type to be solved:
 if *itype* = 1, the problem type is $A*x = \lambda*B*x$;
 if *itype* = 2, the problem type is $A*B*x = \lambda*x$;
 if *itype* = 3, the problem type is $B*A*x = \lambda*x$.

jobz Must be 'N' or 'V'.
 If *jobz* = 'N', then compute eigenvalues only.

	If <code>jobz = 'V'</code> , then compute eigenvalues and eigenvectors.
<code>uplo</code>	Must be 'U' or 'L'.
	If <code>uplo = 'U'</code> , arrays <code>a</code> and <code>b</code> store the upper triangles of <code>A</code> and <code>B</code> ;
	If <code>uplo = 'L'</code> , arrays <code>a</code> and <code>b</code> store the lower triangles of <code>A</code> and <code>B</code> .
<code>n</code>	The order of the matrices <code>A</code> and <code>B</code> ($n \geq 0$).
<code>a, b</code>	Arrays: <code>a</code> (size at least $\max(1, lda * n)$) contains the upper or lower triangle of the Hermitian matrix <code>A</code> , as specified by <code>uplo</code> . <code>b</code> (size at least $\max(1, ldb * n)$) contains the upper or lower triangle of the Hermitian positive definite matrix <code>B</code> , as specified by <code>uplo</code> .
<code>lda</code>	The leading dimension of <code>a</code> ; at least $\max(1, n)$.
<code>ldb</code>	The leading dimension of <code>b</code> ; at least $\max(1, n)$.

Output Parameters

<code>a</code>	On exit, if <code>jobz = 'V'</code> , then if <code>info = 0</code> , <code>a</code> contains the matrix <code>Z</code> of eigenvectors. The eigenvectors are normalized as follows: if <code>itype = 1</code> or <code>2</code> , $Z^H * B * Z = I$; if <code>itype = 3</code> , $Z^H * \text{inv}(B) * Z = I$; If <code>jobz = 'N'</code> , then on exit the upper triangle (if <code>uplo = 'U'</code>) or the lower triangle (if <code>uplo = 'L'</code>) of <code>A</code> , including the diagonal, is destroyed.
<code>b</code>	On exit, if <code>info ≤ n</code> , the part of <code>b</code> containing the matrix is overwritten by the triangular factor <code>U</code> or <code>L</code> from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.
<code>w</code>	Array, size at least $\max(1, n)$. If <code>info = 0</code> , contains the eigenvalues in ascending order.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info > 0`, `cpotrf/zpotrf` or `cheev/zheev` return an error code:

If `info = i ≤ n`, `cheev/zheev` fails to converge, and *i* off-diagonal elements of an intermediate tridiagonal do not converge to zero;

If `info = n + i`, for $1 \leq i \leq n$, then the leading minor of order *i* of `B` is not positive-definite. The factorization of `B` can not be completed and no eigenvalues or eigenvectors are computed.

?sygvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

Syntax

```
lapack_int LAPACKE_ssygvd (int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float* w);
```

```
lapack_int LAPACKE_dsygvd (int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double* w);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A*x = \lambda*B*x, \quad A*B*x = \lambda*x, \quad \text{or} \quad B*A*x = \lambda*x.$$

Here A and B are assumed to be symmetric and B is also positive definite.

It uses a divide and conquer algorithm.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>itype</i>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of A and B .
<i>n</i>	The order of the matrices A and B ($n \geq 0$).
<i>a, b</i>	Arrays: <i>a</i> (size at least $lda*n$) contains the upper or lower triangle of the symmetric matrix A , as specified by <i>uplo</i> . <i>b</i> (size at least $ldb*n$) contains the upper or lower triangle of the symmetric positive definite matrix B , as specified by <i>uplo</i> .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	<p>On exit, if <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>a</i> contains the matrix <i>Z</i> of eigenvectors. The eigenvectors are normalized as follows:</p> <p>if <i>itype</i> = 1 or 2, $Z^T * B * Z = I$;</p> <p>if <i>itype</i> = 3, $Z^T * \text{inv}(B) * Z = I$;</p> <p>If <i>jobz</i> = 'N', then on exit the upper triangle (if <i>uplo</i> = 'U') or the lower triangle (if <i>uplo</i> = 'L') of <i>A</i>, including the diagonal, is destroyed.</p>
<i>b</i>	<p>On exit, if <i>info</i> ≤ <i>n</i>, the part of <i>b</i> containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.</p>
<i>w</i>	<p>Array, size at least max(1, <i>n</i>).</p> <p>If <i>info</i> = 0, contains the eigenvalues in ascending order.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, an error code is returned as specified below.

- For *info* ≤ *n*:
 - If *info* = *i* and *jobz* = 'N', then the algorithm failed to converge; *i* off-diagonal elements of an intermediate tridiagonal form did not converge to zero.
 - If *jobz* = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns *info*/(*n*+1) through mod(*info*, *n*+1).
- For *info* > *n*:
 - If *info* = *n* + *i*, for 1 ≤ *i* ≤ *n*, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

?hegv

Computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem using a divide and conquer method.

Syntax

```
lapack_int LAPACKChegv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b,
lapack_int ldb, float* w );

lapack_int LAPACKZhegv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b,
lapack_int ldb, double* w );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, \quad A^*B^*x = \lambda^*x, \quad \text{or} \quad B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian and B is also positive definite.

It uses a divide and conquer algorithm.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>itype</code>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <code>itype</code> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <code>itype</code> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <code>itype</code> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<code>jobz</code>	Must be 'N' or 'V'. If <code>jobz</code> = 'N', then compute eigenvalues only. If <code>jobz</code> = 'V', then compute eigenvalues and eigenvectors.
<code>uplo</code>	Must be 'U' or 'L'. If <code>uplo</code> = 'U', arrays a and b store the upper triangles of A and B ; If <code>uplo</code> = 'L', arrays a and b store the lower triangles of A and B .
<code>n</code>	The order of the matrices A and B ($n \geq 0$).
<code>a, b</code>	Arrays: a (size at least $\max(1, lda*n)$) contains the upper or lower triangle of the Hermitian matrix A , as specified by <code>uplo</code> . b (size at least $\max(1, ldb*n)$) contains the upper or lower triangle of the Hermitian positive definite matrix B , as specified by <code>uplo</code> .
<code>lda</code>	The leading dimension of a ; at least $\max(1, n)$.
<code>ldb</code>	The leading dimension of b ; at least $\max(1, n)$.

Output Parameters

<code>a</code>	On exit, if <code>jobz</code> = 'V', then if <code>info</code> = 0, a contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if <code>itype</code> = 1 or 2, $Z^H * B * Z = I$; if <code>itype</code> = 3, $Z^H * \text{inv}(B) * Z = I$; If <code>jobz</code> = 'N', then on exit the upper triangle (if <code>uplo</code> = 'U') or the lower triangle (if <code>uplo</code> = 'L') of A , including the diagonal, is destroyed.
----------------	---

b	On exit, if $info \leq n$, the part of b containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.
w	Array, size at least $\max(1, n)$. If $info = 0$, contains the eigenvalues in ascending order.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $jobz = 'N'$, then the algorithm failed to converge; i off-diagonal elements of an intermediate tridiagonal form did not converge to zero;

if $info = i$, and $jobz = 'V'$, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $info/(n+1)$ through $\text{mod}(info, n+1)$.

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
lapack_int LAPACK_essygvx (int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float vl,
float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w, float* z,
lapack_int ldz, lapack_int* ifail);
```

```
lapack_int LAPACK_dsygvx (int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double
vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w,
double* z, lapack_int ldz, lapack_int* ifail);
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A * x = \lambda * B * x, A * B * x = \lambda * x, \text{ or } B * A * x = \lambda * x.$$

Here A and B are assumed to be symmetric and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>itype</i>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>a, b</i>	Arrays: <i>a</i> (size at least $\max(1, lda*n)$) contains the upper or lower triangle of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> . <i>b</i> (size at least $\max(1, ldb*n)$) contains the upper or lower triangle of the symmetric positive definite matrix <i>B</i> , as specified by <i>uplo</i> .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; at least $\max(1, n)$.
<i>vl, vu</i>	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.

*abstol**ldz*

The leading dimension of the output array *z*. Constraints:

$ldz \geq 1$; if *jobz* = 'V', $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout .

Output Parameters

a

On exit, the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is overwritten.

b

On exit, if $info \leq n$, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.

m

The total number of eigenvalues found,

$0 \leq m \leq n$. If *range* = 'A', $m = n$, and if *range* = 'I',
 $m = iu - il + 1$.

w, z

Arrays:

w, size at least $\max(1, n)$.

The first *m* elements of *w* contain the selected eigenvalues in ascending order.

z (size at least $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout) .

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*[*i* - 1]. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^T * B * Z = I$;

if *itype* = 3, $Z^T * \text{inv}(B) * Z = I$;

If *jobz* = 'N', then *z* is not referenced.

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ifail

Array, size at least $\max(1, n)$.

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

Return Values

This function returns a value *info*.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info > 0`, `spotrf/dpotrf` and `ssyevx/dsyevx` returned an error code:

If `info = i ≤ n`, `ssyevx/dsyevx` failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array `ifail`;

If `info = n + i`, for $1 ≤ i ≤ n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If `abstol` is less than or equal to zero, then $\epsilon \|T\|_1$ is used as tolerance, where T is the tridiagonal matrix obtained by reducing C to tridiagonal form, where C is the symmetric matrix of the standard symmetric problem to which the generalized problem is transformed. Eigenvalues will be computed most accurately when `abstol` is set to twice the underflow threshold $2 * \text{?lamch}('S')$, not zero.

If this routine returns with `info > 0`, indicating that some eigenvectors did not converge, set `abstol` to $2 * \text{?lamch}('S')$.

?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

```
lapack_int LAPACKC_chegvx( int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float*
b, lapack_int ldb, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz, lapack_int* ifail );

lapack_int LAPACKC_zhegvx( int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, lapack_complex_double* a, lapack_int lda,
lapack_complex_double* b, lapack_int ldb, double vl, double vu, lapack_int il,
lapack_int iu, double abstol, lapack_int* m, double* w, lapack_complex_double* z,
lapack_int ldz, lapack_int* ifail );
```

Include Files

- `mk1.h`

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda B^*x, \quad A^*B^*x = \lambda^*x, \quad \text{or} \quad B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>itype</i>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>a, b</i>	Arrays: <i>a</i> (size at least $\max(1, lda*n)$) contains the upper or lower triangle of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . <i>b</i> (size at least $\max(1, ldb*n)$) contains the upper or lower triangle of the Hermitian positive definite matrix <i>B</i> , as specified by <i>uplo</i> .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	The leading dimension of <i>b</i> ; at least $\max(1, n)$.
<i>vl, vu</i>	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.

<i>abstol</i>	The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	The leading dimension of the output array <i>z</i> . Constraints: $ldz \geq 1$; if <i>jobz</i> = 'V', $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout.

Output Parameters

<i>a</i>	On exit, the upper triangle (if <i>uplo</i> = 'U') or the lower triangle (if <i>uplo</i> = 'L') of <i>A</i> , including the diagonal, is overwritten.
<i>b</i>	On exit, if <i>info</i> ≤ <i>n</i> , the part of <i>b</i> containing the matrix is overwritten by the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.
<i>m</i>	The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.
<i>w</i>	Array, size at least $\max(1, n)$. The first <i>m</i> elements of <i>w</i> contain the selected eigenvalues in ascending order.
<i>z</i>	Array <i>z</i> (size at least $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout). If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> [<i>i</i> - 1]. The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^H * B * Z = I$; if <i>itype</i> = 3, $Z^H * \text{inv}(B) * Z = I$; If <i>jobz</i> = 'N', then <i>z</i> is not referenced. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i> ; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>ifail</i>	Array, size at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, the <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.

Return Values

This function returns a value *info*.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info > 0`, `cpotrf/zpotrf` and `cheevx/zheevx` returned an error code:

If `info = i ≤ n`, `cheevx/zheevx` failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array `ifail`;

If `info = n + i`, for $1 ≤ i ≤ n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If `abstol` is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing C to tridiagonal form, where C is the symmetric matrix of the standard symmetric problem to which the generalized problem is transformed. Eigenvalues will be computed most accurately when `abstol` is set to twice the underflow threshold $2 * \text{?lamch}('S')$, not zero.

If this routine returns with `info > 0`, indicating that some eigenvectors did not converge, try setting `abstol` to $2 * \text{?lamch}('S')$.

?spgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

```
lapack_int LAPACKE_sspgv (int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, float* ap, float* bp, float* w, float* z, lapack_int ldz);
```

```
lapack_int LAPACKE_dspgv (int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, double* ap, double* bp, double* w, double* z, lapack_int ldz);
```

Include Files

- `mkl.h`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A*x = \lambda*B*x, \quad A*B*x = \lambda*x, \quad \text{or} \quad B*A*x = \lambda*x.$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>itype</code>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <code>itype = 1</code> , the problem type is $A*x = \lambda*B*x$; if <code>itype = 2</code> , the problem type is $A*B*x = \lambda*x$;

	if $itype = 3$, the problem type is $B^*A*x = \lambda x$.
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ap, bp</i>	Arrays: <i>ap</i> contains the packed upper or lower triangle of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i> contains the packed upper or lower triangle of the symmetric matrix <i>B</i> , as specified by <i>uplo</i> . The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$.
<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as <i>B</i> .
<i>w, z</i>	Arrays: <i>w</i> , size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order. <i>z</i> (size $\max(1, ldz*n)$) . If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors. The eigenvectors are normalized as follows: if $itype = 1$ or 2 , $Z^T * B * Z = I$; if $itype = 3$, $Z^T * \text{inv}(B) * Z = I$; If <i>jobz</i> = 'N', then <i>z</i> is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, *spptf/dpptf* and *sspev/dspev* returned an error code:

If $info = i \leq n$, $sspev/dspev$ failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?hpgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with matrices in packed storage.

Syntax

```
lapack_int LAPACKE_chpgv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_float* ap, lapack_complex_float* bp, float* w,
lapack_complex_float* z, lapack_int ldz );
```

```
lapack_int LAPACKE_zhpgv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_double* ap, lapack_complex_double* bp, double* w,
lapack_complex_double* z, lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>itype</i>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of A and B .
<i>n</i>	The order of the matrices A and B ($n \geq 0$).

<i>ap</i> , <i>bp</i>	<p>Arrays:</p> <p><i>ap</i> contains the packed upper or lower triangle of the Hermitian matrix <i>A</i>, as specified by <i>uplo</i>.</p> <p>The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.</p> <p><i>bp</i> contains the packed upper or lower triangle of the Hermitian matrix <i>B</i>, as specified by <i>uplo</i>.</p> <p>The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$.</p>
<i>ldz</i>	<p>The leading dimension of the output array <i>z</i>; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.</p>

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as <i>B</i> .
<i>w</i>	<p>Array, size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the eigenvalues in ascending order.</p>
<i>z</i>	<p>Array <i>z</i> (size $\max(1, ldz*n)$).</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors. The eigenvectors are normalized as follows:</p> <p>if <i>itype</i> = 1 or 2, $Z^H * B * Z = I$;</p> <p>if <i>itype</i> = 3, $Z^H * \text{inv}(B) * Z = I$;</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, *cpptrf*/*zpptrf* and *chpev*/*zhpev* returned an error code:

If *info* = *i* ≤ *n*, *chpev*/*zhpev* failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = *n* + *i*, for $1 \leq i \leq n$, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

?spgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage using a divide and conquer method.

Syntax

```
lapack_int LAPACKE_sspgvd (int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, float* ap, float* bp, float* w, float* z, lapack_int ldz);
```

```
lapack_int LAPACKE_dspgvd (int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, double* ap, double* bp, double* w, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A*x = \lambda*B*x, \quad A*B*x = \lambda*x, \quad \text{or} \quad B*A*x = \lambda*x.$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>itype</i>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of A and B .
<i>n</i>	The order of the matrices A and B ($n \geq 0$).
<i>ap</i> , <i>bp</i>	Arrays: <i>ap</i> contains the packed upper or lower triangle of the symmetric matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i> contains the packed upper or lower triangle of the symmetric matrix B , as specified by <i>uplo</i> . The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$.
<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.

Output Parameters

ap On exit, the contents of *ap* are overwritten.

<i>bp</i>	On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as B .
<i>w, z</i>	<p>Arrays:</p> <p>w, size at least $\max(1, n)$.</p> <p>If $info = 0$, contains the eigenvalues in ascending order.</p> <p>z (size at least $\max(1, ldz * n)$).</p> <p>If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:</p> <p>if $itype = 1$ or 2, $Z^T * B * Z = I$;</p> <p>if $itype = 3$, $Z^T * \text{inv}(B) * Z = I$;</p> <p>If $jobz = 'N'$, then z is not referenced.</p>

Return Values

This function returns a value *info*.

If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info > 0$, `spstrf/dppstrf` and `sspevd/dspevd` returned an error code:

If $info = i \leq n$, `sspevd/dspevd` failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?hpgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with matrices in packed storage using a divide and conquer method.

Syntax

```
lapack_int LAPACKChpgvd( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_float* ap, lapack_complex_float* bp, float* w,
lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKzhpgvd( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_double* ap, lapack_complex_double* bp, double* w,
lapack_complex_double* z, lapack_int ldz );
```

Include Files

- `mkl.h`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$A * x = \lambda * B * x$, $A * B * x = \lambda * x$, or $B * A * x = \lambda * x$.

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>itype</i>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda x$.
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of A and B .
<i>n</i>	The order of the matrices A and B ($n \geq 0$).
<i>ap, bp</i>	Arrays: <i>ap</i> contains the packed upper or lower triangle of the Hermitian matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i> contains the packed upper or lower triangle of the Hermitian matrix B , as specified by <i>uplo</i> . The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$.
<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as B .
<i>w</i>	Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	Array <i>z</i> (size at least $\max(1, ldz*n)$). If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^H * B * Z = I$; if <i>itype</i> = 3, $Z^H * \text{inv}(B) * Z = I$;

If `jobz = 'N'`, then `z` is not referenced.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info > 0`, `cpptrf/zpptrf` and `chpevd/zhpevd` returned an error code:

If `info = i ≤ n`, `chpevd/zhpevd` failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If `info = n + i`, for $1 ≤ i ≤ n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?spgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

```
lapack_int LAPACK_esspgvx (int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, float* ap, float* bp, float vl, float vu, lapack_int il,
lapack_int iu, float abstol, lapack_int* m, float* w, float* z, lapack_int ldz,
lapack_int* ifail);
```

```
lapack_int LAPACK_dspgvx (int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, double* ap, double* bp, double vl, double vu, lapack_int il,
lapack_int iu, double abstol, lapack_int* m, double* w, double* z, lapack_int ldz,
lapack_int* ifail);
```

Include Files

- `mkl.h`

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite.

Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>itype</code>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <code>itype = 1</code> , the problem type is $A^*x = \lambda B^*x$; if <code>itype = 2</code> , the problem type is $A^*B^*x = \lambda^*x$;

	if <i>itype</i> = 3, the problem type is $B^*A*x = \lambda x$.
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ap, bp</i>	Arrays: <i>ap</i> contains the packed upper or lower triangle of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> . The size of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i> contains the packed upper or lower triangle of the symmetric matrix <i>B</i> , as specified by <i>uplo</i> . The size of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$.
<i>vl, vu</i>	If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	The leading dimension of the output array <i>z</i> . Constraints: $ldz \geq 1$; if <i>jobz</i> = 'V', $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout.

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as <i>B</i> .
<i>m</i>	The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.
<i>w</i> , <i>z</i>	Arrays: <i>w</i> , size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order. <i>z</i> (size at least $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout) . If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>). The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^T * B * Z = I$; if <i>itype</i> = 3, $Z^T * \text{inv}(B) * Z = I$; If <i>jobz</i> = 'N', then <i>z</i> is not referenced. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i> ; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>ifail</i>	Array, size at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, the <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, *spptf*/*dpptf* and *sspevx*/*dspevx* returned an error code:

If *info* = *i* ≤ *n*, *sspevx*/*dspevx* failed to converge, and *i* eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If *info* = *n* + *i*, for $1 \leq i \leq n$, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon \|T\|_1$ is used instead, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues are computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, set $abstol$ to $2 * \text{lamch}('S')$.

?hpgvx

Computes selected eigenvalues and, optionally, eigenvectors of a generalized Hermitian positive-definite eigenproblem with matrices in packed storage.

Syntax

```
lapack_int LAPACKE_chpgvx( int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, lapack_complex_float* ap, lapack_complex_float* bp, float vl,
float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w,
lapack_complex_float* z, lapack_int ldz, lapack_int* ifail );
```

```
lapack_int LAPACKE_zhpgvx( int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, lapack_complex_double* ap, lapack_complex_double* bp, double vl,
double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w,
lapack_complex_double* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>itype</i>	Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	Must be 'N' or 'V'.

	<p>If <i>jobz</i> = 'N', then compute eigenvalues only.</p> <p>If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.</p>
<i>range</i>	<p>Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval:</p> $vl < w[i] \leq vu.$ <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p>
<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i>;</p> <p>If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i>.</p>
<i>n</i>	The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ap, bp</i>	<p>Arrays:</p> <p><i>ap</i> contains the packed upper or lower triangle of the Hermitian matrix <i>A</i>, as specified by <i>uplo</i>.</p> <p>The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.</p> <p><i>bp</i> contains the packed upper or lower triangle of the Hermitian matrix <i>B</i>, as specified by <i>uplo</i>.</p> <p>The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$.</p>
<i>vl, vu</i>	<p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $vl < vu$.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il, iu</i>	<p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	<p>The absolute error tolerance for the eigenvalues.</p> <p>See <i>Application Notes</i> for more information.</p>
<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout.

Output Parameters

ap On exit, the contents of *ap* are overwritten.

<i>bp</i>	On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as B .
<i>m</i>	The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.
<i>w</i>	Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	Array <i>z</i> (size at least $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout). If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first m columns of <i>z</i> contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of <i>z</i> holding the eigenvector associated with $w(i)$. The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^H * B * Z = I$; if <i>itype</i> = 3, $Z^H * \text{inv}(B) * Z = I$; If <i>jobz</i> = 'N', then <i>z</i> is not referenced. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i> ; if <i>range</i> = 'V', the exact value of m is not known in advance and an upper bound must be used.
<i>ifail</i>	Array, size at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first m elements of <i>ifail</i> are zero; if <i>info</i> > 0, the <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = - i , the i -th parameter had an illegal value.

If *info* > 0, *cpptrf*/*zpptf* and *chpevx*/*zhpevx* returned an error code:

If *info* = $i \leq n$, *chpevx*/*zhpevx* failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If *info* = $n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * ||T||_1$ is used as tolerance, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{?lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * \text{?lamch}('S')$.

?sbgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

```
lapack_int LAPACKE_ssbgv (int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, float* ab, lapack_int ldab, float* bb, lapack_int ldbb,
float* w, float* z, lapack_int ldz);

lapack_int LAPACKE_dsbgv (int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, double* ab, lapack_int ldab, double* bb, lapack_int ldbb,
double* w, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A*x = \lambda*B*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of A and B .
<i>n</i>	The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	The number of super- or sub-diagonals in B ($kb \geq 0$).
<i>ab, bb</i>	Arrays:

ab (size at least $\max(1, ldab*n)$ for column major layout and $\max(1, ldab*(ka + 1))$ for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix *A* (as specified by *uplo*) in band storage format.

bb (size at least $\max(1, ldbb*n)$ for column major layout and $\max(1, ldbb*(kb + 1))$ for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix *B* (as specified by *uplo*) in band storage format.

ldab

The leading dimension of the array *ab*; must be at least *ka*+1 for column major layout and at least $\max(1, n)$ for row major layout .

ldbb

The leading dimension of the array *bb*; must be at least *kb*+1 for column major layout and at least $\max(1, n)$ for row major layout.

ldz

The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', $ldz \geq \max(1, n)$.

Output Parameters

ab

On exit, the contents of *ab* are overwritten.

bb

On exit, contains the factor *S* from the split Cholesky factorization $B = S^T * S$, as returned by [pbstf/pbstf](#).

w, z

Arrays:

w, size at least $\max(1, n)$.

If *info* = 0, contains the eigenvalues in ascending order.

z (size at least $\max(1, ldz*n)$) .

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that $Z^T * B * Z = I$.

If *jobz* = 'N', then *z* is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, and

if $i \leq n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if *info* = *n* + *i*, for $1 \leq i \leq n$, then [pbstf/pbstf](#) returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

?hbgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.

Syntax

```
lapack_int LAPACKE_chbgv( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab,
lapack_complex_float* bb, lapack_int ldbb, float* w, lapack_complex_float* z,
lapack_int ldz );
```

```
lapack_int LAPACKE_zhbgv( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int ldab,
lapack_complex_double* bb, lapack_int ldbb, double* w, lapack_complex_double* z,
lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here A and B are Hermitian and banded matrices, and matrix B is also positive definite.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of A and B .
<i>n</i>	The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	The number of super- or sub-diagonals in B ($kb \geq 0$).
<i>ab, bb</i>	Arrays: <i>ab</i> (size at least $\max(1, ldab*n)$ for column major layout and $\max(1, ldab*(ka + 1))$ for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by <i>uplo</i>) in band storage format. <i>bb</i> (size at least $\max(1, ldbb*n)$ for column major layout and $\max(1, ldbb*(kb + 1))$ for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix B (as specified by <i>uplo</i>) in band storage format.

<i>ldab</i>	The leading dimension of the array <i>ab</i> ; must be at least $ka+1$ for column major layout and at least $\max(1, n)$ for row major layout.
<i>ldbb</i>	The leading dimension of the array <i>bb</i> ; must be at least $kb+1$ for column major layout and at least $\max(1, n)$ for row major layout.
<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.

Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>S</i> from the split Cholesky factorization $B = S^H * S$, as returned by pbstf/pbstf .
<i>w</i>	Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	Array <i>z</i> (size at least $\max(1, ldz*n)$). If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>). The eigenvectors are normalized so that $Z^H * B * Z = I$. If <i>jobz</i> = 'N', then <i>z</i> is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, and

if $i \leq n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if *info* = *n* + *i*, for $1 \leq i \leq n$, then [pbstf/pbstf](#) returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

?sbgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
lapack_int LAPACKESsbgvd (int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, float* ab, lapack_int ldab, float* bb, lapack_int ldbb,
float* w, float* z, lapack_int ldz);
```

```
lapack_int LAPACKESsbgvd (int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, double* ab, lapack_int ldab, double* bb, lapack_int ldbb,
double* w, double* z, lapack_int ldz);
```

Include Files

- `mk1.h`

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda^*B^*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobz</code>	Must be 'N' or 'V'. If <code>jobz</code> = 'N', then compute eigenvalues only. If <code>jobz</code> = 'V', then compute eigenvalues and eigenvectors.
<code>uplo</code>	Must be 'U' or 'L'. If <code>uplo</code> = 'U', arrays <code>ab</code> and <code>bb</code> store the upper triangles of A and B ; If <code>uplo</code> = 'L', arrays <code>ab</code> and <code>bb</code> store the lower triangles of A and B .
<code>n</code>	The order of the matrices A and B ($n \geq 0$).
<code>ka</code>	The number of super- or sub-diagonals in A ($ka \geq 0$).
<code>kb</code>	The number of super- or sub-diagonals in B ($kb \geq 0$).
<code>ab, bb</code>	Arrays: <code>ab</code> (size at least $\max(1, ldab*n)$ for column major layout and $\max(1, ldab*(ka + 1))$ for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by <code>uplo</code>) in band storage format. <code>bb</code> (size at least $\max(1, ldbb*n)$ for column major layout and $\max(1, ldbb*(kb + 1))$ for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix B (as specified by <code>uplo</code>) in band storage format.
<code>ldab</code>	The leading dimension of the array <code>ab</code> ; must be at least $ka+1$ for column major layout and at least $\max(1, n)$ for row major layout.
<code>ldbb</code>	The leading dimension of the array <code>bb</code> ; must be at least $kb+1$ for column major layout and at least $\max(1, n)$ for row major layout.
<code>ldz</code>	The leading dimension of the output array <code>z</code> ; $ldz \geq 1$. If <code>jobz</code> = 'V', $ldz \geq \max(1, n)$.

Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>S</i> from the split Cholesky factorization $B = S^T * S$, as returned by pbstf/pbstf .
<i>w, z</i>	<p>Arrays:</p> <p><i>w</i>, size at least $\max(1, n)$.</p> <p>If <i>info</i> = 0, contains the eigenvalues in ascending order.</p> <p><i>z</i> (size at least $\max(1, ldz * n)$).</p> <p>If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors, with the <i>i</i>-th column of <i>z</i> holding the eigenvector associated with $w[i - 1]$. The eigenvectors are normalized so that $Z^T * B * Z = I$.</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, and

if $i \leq n$, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if *info* = *n* + *i*, for $1 \leq i \leq n$, then [pbstf/pbstf](#) returned *info* = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

?hbgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```
lapack_int LAPACKChbgvd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab,
lapack_complex_float* bb, lapack_int ldbb, float* w, lapack_complex_float* z,
lapack_int ldz );
```

```
lapack_int LAPACKzhbgvd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int ldab,
lapack_complex_double* bb, lapack_int ldbb, double* w, lapack_complex_double* z,
lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of A and B .
<i>n</i>	The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	The number of super- or sub-diagonals in B ($kb \geq 0$).
<i>ab, bb</i>	Arrays: <i>ab</i> (size at least $\max(1, ldab*n)$ for column major layout and $\max(1, ldab*(ka + 1))$ for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by <i>uplo</i>) in band storage format. <i>bb</i> (size at least $\max(1, ldbb*n)$ for column major layout and $\max(1, ldbb*(kb + 1))$ for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix B (as specified by <i>uplo</i>) in band storage format.
<i>ldab</i>	The leading dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>ldbb</i>	The leading dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.

Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor S from the split Cholesky factorization $B = S^H S$, as returned by pbstf/pbstf .
<i>w</i>	Array, size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	Array <i>z</i> (size at least $\max(1, ldz*n)$).

If `jobz = 'V'`, then if `info = 0`, `z` contains the matrix Z of eigenvectors, with the i -th column of `z` holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^H B Z = I$.

If `jobz = 'N'`, then `z` is not referenced.

Return Values

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info > 0`, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if `info = n + i`, for $1 \leq i \leq n$, then `pbstf/pbstf` returned `info = i` and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?sbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

```
lapack_int LAPACK_?ssbgvx (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int ka, lapack_int kb, float* ab, lapack_int ldab, float* bb,
lapack_int ldbb, float* q, lapack_int ldq, float vl, float vu, lapack_int il, lapack_int
iu, float abstol, lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* ifail);

lapack_int LAPACK_?dsbgvx (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int ka, lapack_int kb, double* ab, lapack_int ldab, double* bb,
lapack_int ldbb, double* q, lapack_int ldq, double vl, double vu, lapack_int il,
lapack_int iu, double abstol, lapack_int* m, double* w, double* z, lapack_int ldz,
lapack_int* ifail);
```

Include Files

- `mkl.h`

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobz</code>	Must be 'N' or 'V'.

	<p>If <code>jobz = 'N'</code>, then compute eigenvalues only.</p> <p>If <code>jobz = 'V'</code>, then compute eigenvalues and eigenvectors.</p>
<code>range</code>	<p>Must be 'A' or 'V' or 'I'.</p> <p>If <code>range = 'A'</code>, the routine computes all eigenvalues.</p> <p>If <code>range = 'V'</code>, the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$.</p> <p>If <code>range = 'I'</code>, the routine computes eigenvalues in range <code>il</code> to <code>iu</code>.</p>
<code>uplo</code>	<p>Must be 'U' or 'L'.</p> <p>If <code>uplo = 'U'</code>, arrays <code>ab</code> and <code>bb</code> store the upper triangles of A and B;</p> <p>If <code>uplo = 'L'</code>, arrays <code>ab</code> and <code>bb</code> store the lower triangles of A and B.</p>
<code>n</code>	The order of the matrices A and B ($n \geq 0$).
<code>ka</code>	<p>The number of super- or sub-diagonals in A</p> <p>($ka \geq 0$).</p>
<code>kb</code>	The number of super- or sub-diagonals in B ($kb \geq 0$).
<code>ab, bb</code>	<p>Arrays:</p> <p><code>ab</code>(size at least $\max(1, ldab*n)$ for column major layout and $\max(1, ldab*(ka + 1))$ for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by <code>uplo</code>) in band storage format.</p> <p><code>bb</code>(size at least $\max(1, ldbb*n)$ for column major layout and $\max(1, ldbb*(kb + 1))$ for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix B (as specified by <code>uplo</code>) in band storage format.</p>
<code>ldab</code>	The leading dimension of the array <code>ab</code> ; must be at least $ka+1$ for column major layout and at least $\max(1, n)$ for row major layout.
<code>ldbb</code>	The leading dimension of the array <code>bb</code> ; must be at least $kb+1$ for column major layout and at least $\max(1, n)$ for row major layout.
<code>vl, vu</code>	<p>If <code>range = 'V'</code>, the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $vl < vu$.</p> <p>If <code>range = 'A'</code> or 'I', <code>vl</code> and <code>vu</code> are not referenced.</p>
<code>il, iu</code>	<p>If <code>range = 'I'</code>, the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.</p> <p>If <code>range = 'A'</code> or 'V', <code>il</code> and <code>iu</code> are not referenced.</p>

<i>abstol</i>	The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.
<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.
<i>ldq</i>	The leading dimension of the output array <i>q</i> ; $ldq < 1$. If <i>jobz</i> = 'V', $ldq < \max(1, n)$.

Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>S</i> from the split Cholesky factorization $B = S^T * S$, as returned by pbstf / pbstf .
<i>m</i>	The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.
<i>w, z, q</i>	Arrays: <i>w</i> , size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order. <i>z</i> (size $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout) . If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>). The eigenvectors are normalized so that $Z^T * B * Z = I$. If <i>jobz</i> = 'N', then <i>z</i> is not referenced. <i>q</i> (size $\max(1, ldq * n)$) . If <i>jobz</i> = 'V', then <i>q</i> contains the <i>n</i> -by- <i>n</i> matrix used in the reduction of $A * x = \lambda * B * x$ to standard form, that is, $C * x = \lambda * x$ and consequently <i>C</i> to tridiagonal form. If <i>jobz</i> = 'N', then <i>q</i> is not referenced.
<i>ifail</i>	Array, size <i>m</i> . If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, the <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then `pbstf/pbstf` returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon \cdot \|T\|_1$ is used as tolerance, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 * \text{lamch}('S')$.

?hbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.

Syntax

```
lapack_int LAPACKE_chbgvx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab,
lapack_complex_float* bb, lapack_int ldbb, lapack_complex_float* q, lapack_int ldq,
float vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w,
lapack_complex_float* z, lapack_int ldz, lapack_int* ifail );
```

```
lapack_int LAPACKE_zhbgvx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int ldab,
lapack_complex_double* bb, lapack_int ldbb, lapack_complex_double* q, lapack_int ldq,
double vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m,
double* w, lapack_complex_double* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- `mkl.h`

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A*x = \lambda*B*x$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobz</i>	Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.

<i>range</i>	<p>Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the routine computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval:</p> $vl < w[i] \leq vu.$ <p>If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</p>
<i>uplo</i>	<p>Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of <i>A</i> and <i>B</i>;</p> <p>If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of <i>A</i> and <i>B</i>.</p>
<i>n</i>	The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>ka</i>	<p>The number of super- or sub-diagonals in <i>A</i></p> <p>($ka \geq 0$).</p>
<i>kb</i>	The number of super- or sub-diagonals in <i>B</i> ($kb \geq 0$).
<i>ab, bb</i>	<p>Arrays:</p> <p><i>ab</i>(size at least $\max(1, ldab*n)$ for column major layout and $\max(1, ldab*(ka + 1))$ for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format.</p> <p><i>bb</i>(size at least $\max(1, ldbb*n)$ for column major layout and $\max(1, ldbb*(kb + 1))$ for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix <i>B</i> (as specified by <i>uplo</i>) in band storage format.</p>
<i>ldab</i>	The leading dimension of the array <i>ab</i> ; must be at least $ka+1$ for column major layout and at least $\max(1, n)$ for row major layout.
<i>ldbb</i>	The leading dimension of the array <i>bb</i> ; must be at least $kb+1$ for column major layout and at least $\max(1, n)$ for row major layout.
<i>vl, vu</i>	<p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>Constraint: $vl < vu$.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il, iu</i>	<p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.</p> <p>Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	The absolute error tolerance for the eigenvalues. See <i>Application Notes</i> for more information.

<i>ldz</i>	The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$ for column major layout and at least $\max(1, m)$ for row major layout.
<i>ldq</i>	The leading dimension of the output array <i>q</i> ; $ldq \geq 1$. If <i>jobz</i> = 'V', $ldq \geq \max(1, n)$.

Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor <i>S</i> from the split Cholesky factorization $B = S^H * S$, as returned by pbstf/pbstf .
<i>m</i>	The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.
<i>w</i>	Array <i>w</i> , size at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z, q</i>	Arrays: <i>z</i> (size $\max(1, ldz * m)$ for column major layout and $\max(1, ldz * n)$ for row major layout). If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with $w[i - 1]$. The eigenvectors are normalized so that $Z^H * B * Z = I$. If <i>jobz</i> = 'N', then <i>z</i> is not referenced. <i>q</i> (size $\max(1, ldq * n)$). If <i>jobz</i> = 'V', then <i>q</i> contains the <i>n</i> -by- <i>n</i> matrix used in the reduction of $Ax = \lambda Bx$ to standard form, that is, $Cx = \lambda x$ and consequently <i>C</i> to tridiagonal form. If <i>jobz</i> = 'N', then <i>q</i> is not referenced.
<i>ifail</i>	Array, size at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, the <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then [pbstf/pbstf](#) returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon \cdot \max(|a|, |b|)$, where ϵ is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon \cdot ||T||_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{lamch}('S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2 * \text{lamch}('S')$.

Generalized Nonsymmetric Eigenvalue Problems: LAPACK Driver Routines

This topic describes LAPACK driver routines used for solving generalized nonsymmetric eigenproblems. See also [computational routines](#) that can be called to solve these problems. [Table "Driver Routines for Solving Generalized Nonsymmetric Eigenproblems"](#) lists all such driver routines.

Driver Routines for Solving Generalized Nonsymmetric Eigenproblems

Routine Name	Operation performed
gges	Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.
ggesx	Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.
gges3	Computes generalized Schur factorization for a pair of matrices.
ggeev	Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.
ggeevx	Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.
ggeev3	Computes generalized Schur factorization for a pair of matrices.

?gges

Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.

Syntax

```
lapack_int LAPACKE_sgges( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_S_SELECT3 select, lapack_int n, float* a, lapack_int lda, float* b, lapack_int
ldb, lapack_int* sdim, float* alphas, float* alphas, float* beta, float* vs1, lapack_int
ldvs1, float* vsr, lapack_int ldvsr );
```

```
lapack_int LAPACKE_dgges( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_D_SELECT3 select, lapack_int n, double* a, lapack_int lda, double* b, lapack_int
ldb, lapack_int* sdim, double* alphas, double* alphas, double* beta, double* vs1,
lapack_int ldvs1, double* vsr, lapack_int ldvsr );
```

```

lapack_int LAPACKE_cgges( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_C_SELECT2 select, lapack_int n, lapack_complex_float* a, lapack_int lda,
lapack_complex_float* b, lapack_int ldb, lapack_int* sdim, lapack_complex_float* alpha,
lapack_complex_float* beta, lapack_complex_float* vsl, lapack_int ldvsl,
lapack_complex_float* vsr, lapack_int ldvsr );

lapack_int LAPACKE_zgges( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_Z_SELECT2 select, lapack_int n, lapack_complex_double* a, lapack_int lda,
lapack_complex_double* b, lapack_int ldb, lapack_int* sdim, lapack_complex_double*
alpha, lapack_complex_double* beta, lapack_complex_double* vsl, lapack_int ldvsl,
lapack_complex_double* vsr, lapack_int ldvsr );

```

Include Files

- mkl.h

Description

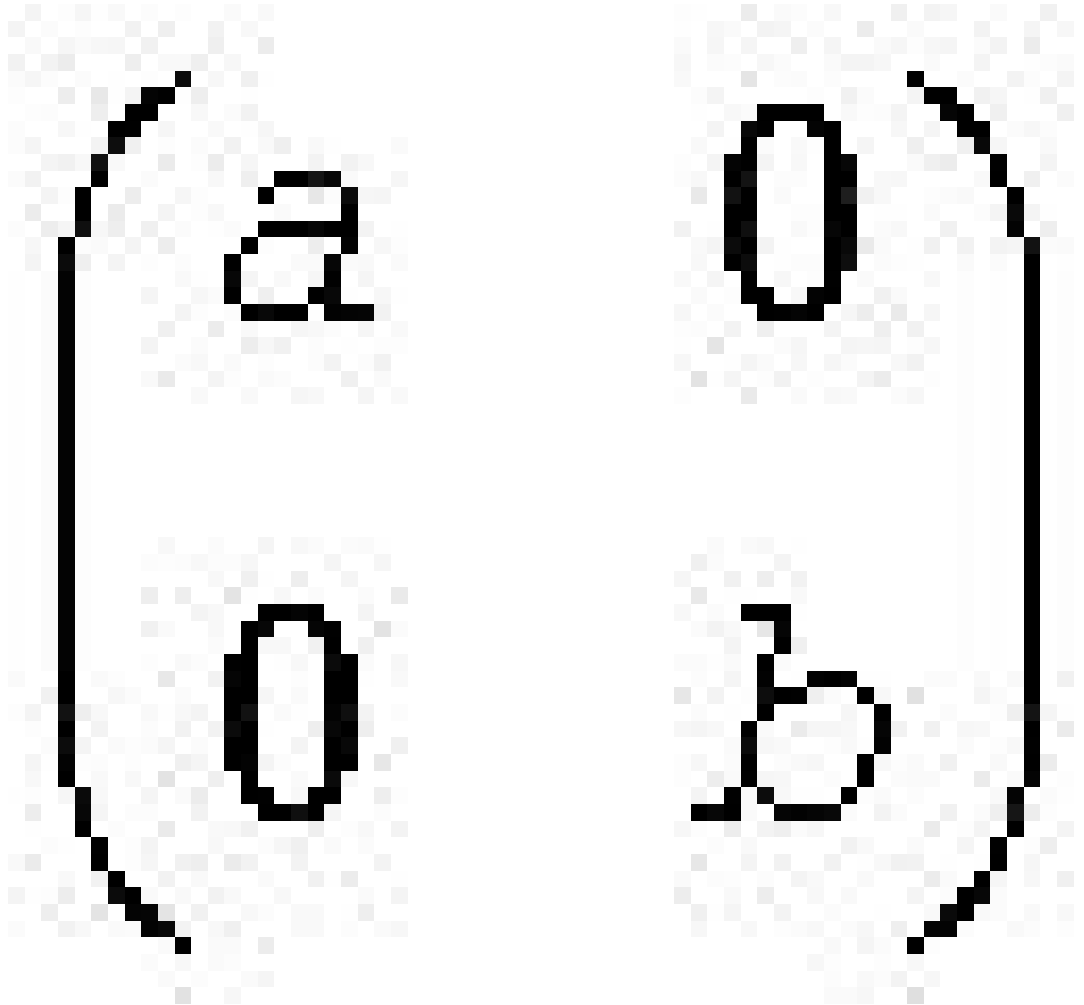
The `?gges` routine computes the generalized eigenvalues, the generalized real/complex Schur form (S,T) , optionally, the left and/or right matrices of Schur vectors (vsl and vsr) for a pair of n -by- n real/complex nonsymmetric matrices (A,B) . This gives the generalized Schur factorization

$$(A,B) = (vsl * S * vsr^H, vsl * T * vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T . The leading columns of vsl and vsr then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

If only the generalized eigenvalues are needed, use the driver [ggeev](#) instead, which is faster.

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio $alpha / beta = w$, such that $A - w*B$ is singular. It is usually represented as the pair $(alpha, beta)$, as there is a reasonable interpretation for $beta=0$ or for both being zero. A pair of matrices (S,T) is in the generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S are "standardized" by making the corresponding elements of T have the form:



and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (S, T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal of T are non-negative real numbers.

The `?gges` routine replaces the deprecated `?gegs` routine.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobvsl</code>	Must be 'N' or 'V'. If <code>jobvsl</code> = 'N', then the left Schur vectors are not computed. If <code>jobvsl</code> = 'V', then the left Schur vectors are computed.
<code>jobvsr</code>	Must be 'N' or 'V'. If <code>jobvsr</code> = 'N', then the right Schur vectors are not computed. If <code>jobvsr</code> = 'V', then the right Schur vectors are computed.
<code>sort</code>	Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.

select

If *sort* = 'N', then eigenvalues are not ordered.

If *sort* = 'S', eigenvalues are ordered (see *select*).

The *select* parameter is a pointer to a function returning a value of `lapack_logical` type. For different flavors the function has different arguments:

```
LAPACKE_sgges: lapack_logical (*LAPACK_S_SELECT3) ( const
float*, const float*, const float* );
```

```
LAPACKE_dgges: lapack_logical (*LAPACK_D_SELECT3) ( const
double*, const double*, const double* );
```

```
LAPACKE_cgges: lapack_logical (*LAPACK_C_SELECT2) ( const
lapack_complex_float*, const lapack_complex_float* );
```

```
LAPACKE_zgges: lapack_logical (*LAPACK_Z_SELECT2) ( const
lapack_complex_double*, const lapack_complex_double* );
```

If *sort* = 'S', *select* is used to select eigenvalues to sort to the top left of the Schur form.

If *sort* = 'N', *select* is not referenced.

For real flavors:

An eigenvalue $(\alpha_{phar}[j] + \alpha_{phai}[j])/beta[j]$ is selected if *select*(*alphar*[*j*], *alphai*[*j*], *beta*[*j*]) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy *select*(*alphar*[*j*], *alphai*[*j*], *beta*[*j*]) = 1 after ordering. In this case *info* is set to *n*+2 .

For complex flavors:

An eigenvalue $\alpha_{pha}[j] / \beta_{ta}[j]$ is selected if *select*(*alpha*[*j*], *beta*[*j*]) is true.

Note that a selected complex eigenvalue may no longer satisfy *select*(*alpha*[*j*], *beta*[*j*]) = 1 after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* is set to *n*+2 (see *info* below).

n

The order of the matrices *A*, *B*, *vs**l*, and *vs**r* (*n* ≥ 0).

a, *b*

Arrays:

a (size at least max(1, *lda***n*)) is an array containing the *n*-by-*n* matrix *A* (first of the pair of matrices).

b (size at least max(1, *ldb***n*)) is an array containing the *n*-by-*n* matrix *B* (second of the pair of matrices).

lda

The leading dimension of the array *a*. Must be at least max(1, *n*).

ldb

The leading dimension of the array *b*. Must be at least max(1, *n*).

*ldvs**l*, *ldvs**r*

The leading dimensions of the output matrices *vs**l* and *vs**r*, respectively. Constraints:

*ldvs**l* ≥ 1. If *jobvs**l* = 'V', *ldvs**l* ≥ max(1, *n*).

$ldvsr \geq 1$. If $jobvsr = 'V'$, $ldvsr \geq \max(1, n)$.

Output Parameters

<i>a</i>	On exit, this array has been overwritten by its generalized Schur form <i>S</i> .
<i>b</i>	On exit, this array has been overwritten by its generalized Schur form <i>T</i> .
<i>sdim</i>	<p>If $sort = 'N'$, $sdim = 0$.</p> <p>If $sort = 'S'$, $sdim$ is equal to the number of eigenvalues (after sorting) for which <i>select</i> is true.</p> <p>Note that for real flavors complex conjugate pairs for which <i>select</i> is true for either eigenvalue count as 2.</p>
<i>alphas</i> , <i>alphai</i>	<p>Arrays, size at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.</p> <p>See <i>beta</i>.</p>
<i>alpha</i>	Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See <i>beta</i> .
<i>beta</i>	<p>Array, size at least $\max(1, n)$.</p> <p>For real flavors:</p> <p>On exit, $(\alpha[j] + \alpha_i[j]*i)/\beta[j]$, $j=0, \dots, n-1$, will be the generalized eigenvalues.</p> <p>$\alpha[j] + \alpha_i[j]*i$ and $\beta[j]$, $j=0, \dots, n-1$ are the diagonals of the complex Schur form (<i>S</i>,<i>T</i>) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (<i>A</i>,<i>B</i>) were further reduced to triangular form using complex unitary transformations. If $\alpha_i[j]$ is zero, then the <i>j</i>-th eigenvalue is real; if positive, then the <i>j</i>-th and (<i>j</i>+1)-st eigenvalues are a complex conjugate pair, with $\alpha_i[j+1]$ negative.</p> <p>For complex flavors:</p> <p>On exit, $\alpha[j]/\beta[j]$, $j=0, \dots, n-1$, will be the generalized eigenvalues. $\alpha[j]$ and $\beta[j]$, $j=0, \dots, n-1$ are the diagonals of the complex Schur form (<i>S</i>,<i>T</i>) output by <i>cgges/zgges</i>. The $\beta[j]$ will be non-negative real.</p> <p>See also <i>Application Notes</i> below.</p>
<i>vs1</i> , <i>vsr</i>	<p>Arrays:</p> <p><i>vs1</i> (size at least $\max(1, ldvs1*n)$).</p> <p>If $jobvs1 = 'V'$, this array will contain the left Schur vectors.</p> <p>If $jobvs1 = 'N'$, <i>vs1</i> is not referenced.</p> <p><i>vsr</i> (size at least $\max(1, ldvsr*n)$).</p> <p>If $jobvsr = 'V'$, this array will contain the right Schur vectors.</p> <p>If $jobvsr = 'N'$, <i>vsr</i> is not referenced.</p>

Return Values

This function returns a value *info*.

If `info=0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = i`, and

`i ≤ n`:

the QZ iteration failed. (*A*, *B*) is not in Schur form, but `alphan[j]`, `alphai[j]` (for real flavors), or `alpha[j]` (for complex flavors), and `beta[j]`, $j = info, \dots, n - 1$ should be correct.

`i > n`: errors that usually indicate LAPACK problems:

`i = n+1`: other than QZ iteration failed in [hgeqz](#);

`i = n+2`: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy `select = 1`. This could also be caused due to scaling;

`i = n+3`: reordering failed in [tgsen](#).

Application Notes

The quotients `alphan[j]/beta[j]` and `alphai[j]/beta[j]` may easily over- or underflow, and `beta[j]` may even be zero. Thus, you should avoid simply computing the ratio. However, `alphan` and `alphai` will be always less than and usually comparable with `norm(A)` in magnitude, and `beta` always less than and usually comparable with `norm(B)`.

?ggesx

Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.

Syntax

```
lapack_int LAPACKE_sggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_S_SELECT3 select, char sense, lapack_int n, float* a, lapack_int lda, float* b,
lapack_int ldb, lapack_int* sdim, float* alphan, float* alphai, float* beta, float* vsl,
lapack_int ldvsl, float* vsr, lapack_int ldvsr, float* rconde, float* rcondv );

lapack_int LAPACKE_dggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_D_SELECT3 select, char sense, lapack_int n, double* a, lapack_int lda, double* b,
lapack_int ldb, lapack_int* sdim, double* alphan, double* alphai, double* beta, double*
vsl, lapack_int ldvsl, double* vsr, lapack_int ldvsr, double* rconde, double* rcondv );

lapack_int LAPACKE_cggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_C_SELECT2 select, char sense, lapack_int n, lapack_complex_float* a, lapack_int
lda, lapack_complex_float* b, lapack_int ldb, lapack_int* sdim, lapack_complex_float*
alpha, lapack_complex_float* beta, lapack_complex_float* vsl, lapack_int ldvsl,
lapack_complex_float* vsr, lapack_int ldvsr, float* rconde, float* rcondv );

lapack_int LAPACKE_zggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_Z_SELECT2 select, char sense, lapack_int n, lapack_complex_double* a, lapack_int
lda, lapack_complex_double* b, lapack_int ldb, lapack_int* sdim, lapack_complex_double*
alpha, lapack_complex_double* beta, lapack_complex_double* vsl, lapack_int ldvsl,
lapack_complex_double* vsr, lapack_int ldvsr, double* rconde, double* rcondv );
```

Include Files

- `mkl.h`

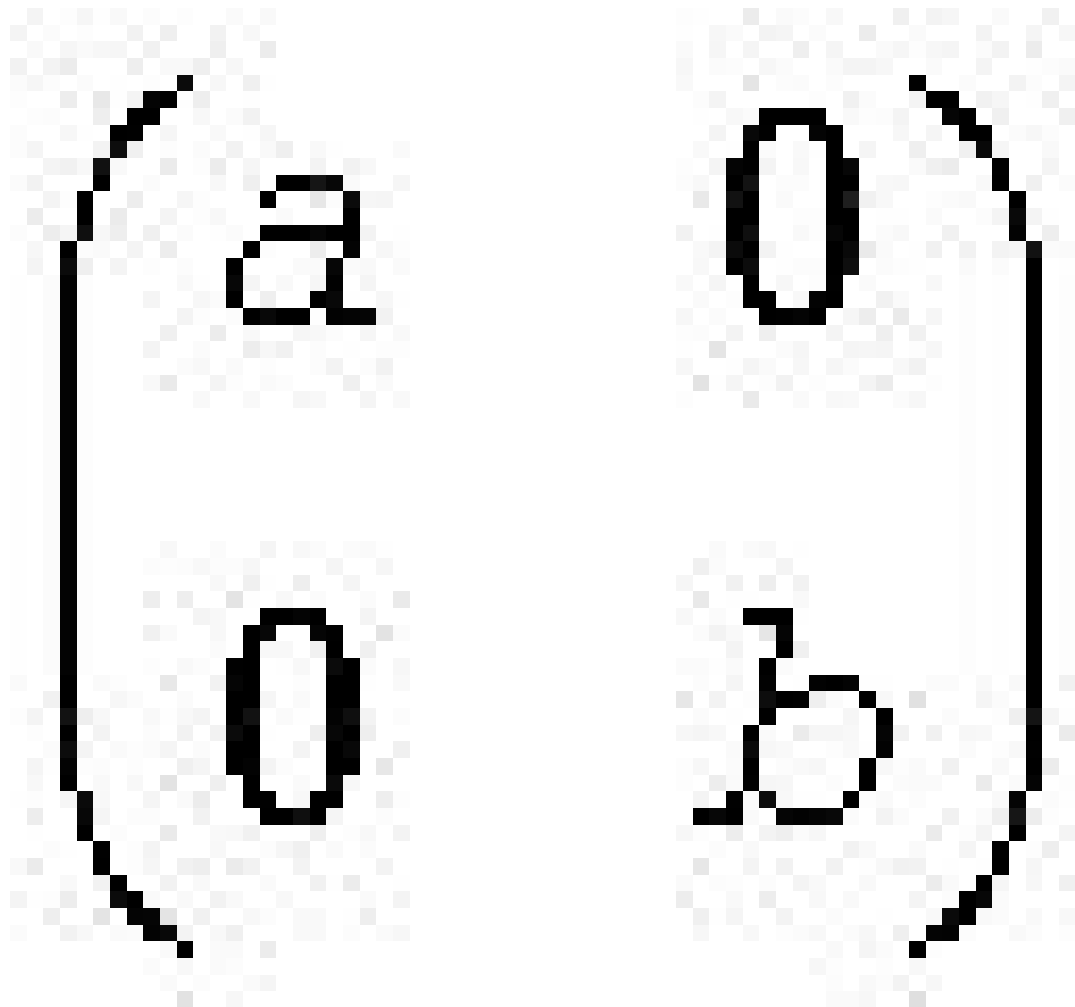
Description

The routine computes for a pair of n -by- n real/complex nonsymmetric matrices (A,B) , the generalized eigenvalues, the generalized real/complex Schur form (S,T) , optionally, the left and/or right matrices of Schur vectors (vs_l and vs_r). This gives the generalized Schur factorization

$$(A,B) = (vs_l * S * vs_r^H, vs_l * T * vs_r^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T ; computes a reciprocal condition number for the average of the selected eigenvalues ($rconde$); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues ($rcondv$). The leading columns of vs_l and vs_r then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio $alpha / beta = w$, such that $A - w*B$ is singular. It is usually represented as the pair $(alpha, beta)$, as there is a reasonable interpretation for $beta=0$ or for both being zero. A pair of matrices (S,T) is in generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S will be "standardized" by making the corresponding elements of T have the form:



and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (S, T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal of T are non-negative real numbers.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobvsl</i>	<p>Must be 'N' or 'V'.</p> <p>If <i>jobvsl</i> = 'N', then the left Schur vectors are not computed.</p> <p>If <i>jobvsl</i> = 'V', then the left Schur vectors are computed.</p>
<i>jobvsr</i>	<p>Must be 'N' or 'V'.</p> <p>If <i>jobvsr</i> = 'N', then the right Schur vectors are not computed.</p> <p>If <i>jobvsr</i> = 'V', then the right Schur vectors are computed.</p>
<i>sort</i>	<p>Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.</p> <p>If <i>sort</i> = 'N', then eigenvalues are not ordered.</p> <p>If <i>sort</i> = 'S', eigenvalues are ordered (see <i>select</i>).</p>
<i>select</i>	<p>The <i>select</i> parameter is a pointer to a function returning a value of <code>lapack_logical</code> type. For different flavors the function has different arguments:</p> <pre>LAPACKE_sggesx: lapack_logical (*LAPACK_S_SELECT3) (const float*, const float*, const float*);</pre> <pre>LAPACKE_dggesx: lapack_logical (*LAPACK_D_SELECT3) (const double*, const double*, const double*);</pre> <pre>LAPACKE_cggesx: lapack_logical (*LAPACK_C_SELECT2) (const lapack_complex_float*, const lapack_complex_float*);</pre> <pre>LAPACKE_zggesx: lapack_logical (*LAPACK_Z_SELECT2) (const lapack_complex_double*, const lapack_complex_double*);</pre> <p>If <i>sort</i> = 'S', <i>select</i> is used to select eigenvalues to sort to the top left of the Schur form.</p> <p>If <i>sort</i> = 'N', <i>select</i> is not referenced.</p> <p>For real flavors:</p> <p>An eigenvalue $(\alpha[j] + \beta[j]i)/\beta[j]$ is selected if <i>select</i>($\alpha[j]$, $\beta[j]$) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.</p> <p>Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy <i>select</i>($\alpha[j]$, $\beta[j]$) = 1 after ordering. In this case <i>info</i> is set to $n+2$.</p> <p>For complex flavors:</p> <p>An eigenvalue $\alpha[j] / \beta[j]$ is selected if <i>select</i>($\alpha[j]$, $\beta[j]$) is true.</p>

Note that a selected complex eigenvalue may no longer satisfy $\text{select}(\alpha[j], \beta[j]) = 1$ after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case *info* is set to $n+2$ (see *info* below).

sense

Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.

If *sense* = 'N', none are computed;

If *sense* = 'E', computed for average of selected eigenvalues only;

If *sense* = 'V', computed for selected deflating subspaces only;

If *sense* = 'B', computed for both.

If *sense* is 'E', 'V', or 'B', then *sort* must equal 'S'.

n

The order of the matrices *A*, *B*, *vsL*, and *vsR* ($n \geq 0$).

a, *b*

Arrays:

a (size at least $\max(1, \text{lda} * n)$) is an array containing the n -by- n matrix *A* (first of the pair of matrices).

b (size at least $\max(1, \text{ldb} * n)$) is an array containing the n -by- n matrix *B* (second of the pair of matrices).

lda

The leading dimension of the array *a*.

Must be at least $\max(1, n)$.

ldb

The leading dimension of the array *b*.

Must be at least $\max(1, n)$.

ldvsl, *ldvsr*

The leading dimensions of the output matrices *vsL* and *vsR*, respectively.
Constraints:

$\text{ldvsl} \geq 1$. If *jobvsl* = 'V', $\text{ldvsl} \geq \max(1, n)$.

$\text{ldvsr} \geq 1$. If *jobvsr* = 'V', $\text{ldvsr} \geq \max(1, n)$.

Output Parameters

a

On exit, this array has been overwritten by its generalized Schur form *S*.

b

On exit, this array has been overwritten by its generalized Schur form *T*.

sdim

If *sort* = 'N', *sdim* = 0.

If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *select* is true.

Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

alphar, *alphai*

Arrays, size at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

<i>alpha</i>	Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See <i>beta</i> .
<i>beta</i>	<p>Array, size at least $\max(1, n)$.</p> <p>For real flavors:</p> <p>On exit, $(\text{alpha}[j] + \text{alpha}[j]*i)/\text{beta}[j]$, $j=0, \dots, n-1$ will be the generalized eigenvalues.</p> <p>$\text{alpha}[j] + \text{alpha}[j]*i$ and $\text{beta}[j]$, $j=0, \dots, n-1$ are the diagonals of the complex Schur form (S, T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A, B) were further reduced to triangular form using complex unitary transformations. If $\text{alpha}[j]$ is zero, then the j-th eigenvalue is real; if positive, then the j-th and $(j+1)$-st eigenvalues are a complex conjugate pair, with $\text{alpha}[j+1]$ negative.</p> <p>For complex flavors:</p> <p>On exit, $\text{alpha}[j]/\text{beta}[j]$, $j=0, \dots, n-1$ will be the generalized eigenvalues. $\text{alpha}[j]$ and $\text{beta}[j]$, $j=0, \dots, n-1$ are the diagonals of the complex Schur form (S, T) output by <code>cggesx/zggesx</code>. The $\text{beta}[j]$ will be non-negative real.</p> <p>See also <i>Application Notes</i> below.</p>
<i>vs1, vsr</i>	<p>Arrays:</p> <p><i>vs1</i> (size at least $\max(1, \text{ldvs1}*n)$).</p> <p>If <i>jobvs1</i> = 'V', this array will contain the left Schur vectors.</p> <p>If <i>jobvs1</i> = 'N', <i>vs1</i> is not referenced.</p> <p><i>vsr</i> (size at least $\max(1, \text{ldvsr}*n)$).</p> <p>If <i>jobvsr</i> = 'V', this array will contain the right Schur vectors.</p> <p>If <i>jobvsr</i> = 'N', <i>vsr</i> is not referenced.</p>
<i>rconde, rcondv</i>	<p>Arrays, size 2 each</p> <p>If <i>sense</i> = 'E' or 'B', <i>rconde</i>(1) and <i>rconde</i>(2) contain the reciprocal condition numbers for the average of the selected eigenvalues.</p> <p>Not referenced if <i>sense</i> = 'N' or 'V'.</p> <p>If <i>sense</i> = 'V' or 'B', <i>rcondv</i>[0] and <i>rcondv</i>[1] contain the reciprocal condition numbers for the selected deflating subspaces.</p> <p>Not referenced if <i>sense</i> = 'N' or 'E'.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and

$i \leq n$:

the QZ iteration failed. (A, B) is not in Schur form, but $\text{alpha}[j]$, $\text{alpha}[j]$ (for real flavors), or $\text{alpha}[j]$ (for complex flavors), and $\text{beta}[j]$, $j = \text{info}, \dots, n-1$ should be correct.

$i > n$: errors that usually indicate LAPACK problems:

$i = n+1$: other than QZ iteration failed in [hgeqz](#);

$i = n+2$: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy $select = 1$. This could also be caused due to scaling;

$i = n+3$: reordering failed in [tgsen](#).

Application Notes

The quotients $alphan[j]/beta[j]$ and $alphai[j]/beta[j]$ may easily over- or underflow, and $beta[j]$ may even be zero. Thus, you should avoid simply computing the ratio. However, $alphan$ and $alphai$ will be always less than and usually comparable with $norm(A)$ in magnitude, and $beta$ always less than and usually comparable with $norm(B)$.

?gges3

Computes generalized Schur factorization for a pair of matrices.

Syntax

```
lapack_int LAPACKE_sgges3 (int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_S_SELECT3 selctg, lapack_int n, float * a, lapack_int lda, float * b, lapack_int
ldb, lapack_int * sdim, float * alphan, float * alphai, float * beta, float * vsl,
lapack_int ldvsl, float * vsr, lapack_int ldvsr);
```

```
lapack_int LAPACKE_dgges3 (int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_D_SELECT3 selctg, lapack_int n, double * a, lapack_int lda, double * b,
lapack_int ldb, lapack_int * sdim, double * alphan, double * alphai, double * beta,
double * vsl, lapack_int ldvsl, double * vsr, lapack_int ldvsr);
```

```
lapack_int LAPACKE_cgges3 (int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_C_SELECT2 selctg, lapack_int n, lapack_complex_float * a, lapack_int lda,
lapack_complex_float * b, lapack_int ldb, lapack_int * sdim, lapack_complex_float *
alpha, lapack_complex_float * beta, lapack_complex_float * vsl, lapack_int ldvsl,
lapack_complex_float * vsr, lapack_int ldvsr);
```

```
lapack_int LAPACKE_zgges3 (int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_Z_SELECT2 selctg, lapack_int n, lapack_complex_double * a, lapack_int lda,
lapack_complex_double * b, lapack_int ldb, lapack_int * sdim, lapack_complex_double *
alpha, lapack_complex_double * beta, lapack_complex_double * vsl, lapack_int ldvsl,
lapack_complex_double * vsr, lapack_int ldvsr);
```

Include Files

- `mk1.h`

Description

For a pair of n -by- n real or complex nonsymmetric matrices (A,B) , `?gges3` computes the generalized eigenvalues, the generalized real or complex Schur form (S,T) , and optionally the left or right matrices of Schur vectors $(VSL$ and $VSR)$. This gives the generalized Schur factorization

$$(A,B) = ((VSL)*S*(VSR)^T, (VSL)*T*(VSR)^T) \text{ for real } (A,B)$$

or

$$(A,B) = ((VSL)*S*(VSR)^H, (VSL)*T*(VSR)^H) \text{ for complex } (A,B)$$

where $(VSR)^H$ is the conjugate-transpose of VSR .

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T . The leading columns of VSL and VSR then form an orthonormal basis for the corresponding left and right eigenspaces (deflating subspaces).

NOTE

If only the generalized eigenvalues are needed, use the driver `?ggeev` instead, which is faster.

A generalized eigenvalue for a pair of matrices (A,B) is a scalar w or a ratio $\alpha/\beta = w$, such that $A - w*B$ is singular. It is usually represented as the pair (α,β) , as there is a reasonable interpretation for $\beta=0$ or both being zero.

For real flavors:

A pair of matrices (S,T) is in generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S will be "standardized" by making the corresponding elements of T have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in S and T have a complex conjugate pair of generalized eigenvalues.

For complex flavors:

A pair of matrices (S,T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal elements of T are non-negative real numbers.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>jobvsl</code>	= 'N': do not compute the left Schur vectors;
<code>jobvsr</code>	= 'N': do not compute the right Schur vectors; = 'V': compute the right Schur vectors.
<code>sort</code>	Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form. = 'N': Eigenvalues are not ordered; = 'S': Eigenvalues are ordered (see <code>selctg</code>).
<code>selctg</code>	<code>selctg</code> is a function of three arguments for real flavors or two arguments for complex flavors. <code>selctg</code> must be declared EXTERNAL in the calling subroutine. If <code>sort = 'N'</code> , <code>selctg</code> is not referenced. If <code>sort = 'S'</code> , <code>selctg</code> is used to select eigenvalues to sort to the top left of the Schur form. For real flavors: An eigenvalue $(\alpha_{\text{phar}}[j - 1] + \alpha_{\text{phai}}[j - 1]i)/\beta[j - 1]$ is selected if <code>selctg($\alpha_{\text{phar}}[j - 1]$, $\alpha_{\text{phai}}[j - 1]$, $\beta[j - 1]$)</code> is true. In other words, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy $\text{selctg}(\text{alphan}[j - 1], \text{alphai}[j - 1], \text{beta}[j - 1]) \neq 0$ after ordering. *info* is to be set to $n+2$ in this case.

For complex flavors:

An eigenvalue $\text{alpha}[j - 1]/\text{beta}[j - 1]$ is selected if $\text{selctg}(\text{alpha}[j - 1], \text{beta}[j - 1])$ is true.

Note that a selected complex eigenvalue may no longer satisfy $\text{selctg}(\text{alpha}[j - 1], \text{beta}[j - 1]) \neq 0$ after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned), in this case *gges3* returns $n + 2$.

<i>n</i>	The order of the matrices <i>A</i> , <i>B</i> , <i>VSL</i> , and <i>VSR</i> . $n \geq 0$.
<i>a</i>	Array, size ($lda*n$). On entry, the first of the pair of matrices.
<i>lda</i>	The leading dimension of <i>a</i> . $lda \geq \max(1, n)$.
<i>b</i>	Array, size ($ldb*n$). On entry, the second of the pair of matrices.
<i>ldb</i>	The leading dimension of <i>b</i> . $ldb \geq \max(1, n)$.
<i>ldvsl</i>	The leading dimension of the matrix <i>VSL</i> . $ldvsl \geq 1$, and if <i>jobvsl</i> = 'V', $ldvsl \geq n$.
<i>ldvsr</i>	The leading dimension of the matrix <i>VSR</i> . $ldvsr \geq 1$, and if <i>jobvsr</i> = 'V', $ldvsr \geq n$.

Output Parameters

<i>a</i>	On exit, <i>a</i> is overwritten by its generalized Schur form <i>S</i> .
<i>b</i>	On exit, <i>b</i> is overwritten by its generalized Schur form <i>T</i> .
<i>sdim</i>	If <i>sort</i> = 'N', <i>sdim</i> = 0. If <i>sort</i> = 'S', <i>sdim</i> = number of eigenvalues (after sorting) for which <i>selctg</i> is true.
<i>alpha</i>	Array, size (<i>n</i>).
<i>alphan</i>	Array, size (<i>n</i>).
<i>alphai</i>	Array, size (<i>n</i>).
<i>beta</i>	Array, size (<i>n</i>).
	For real flavors:
	On exit, $(\text{alphan}[j - 1] + \text{alphai}[j - 1]*i)/\text{beta}[j - 1]$, $j=1, \dots, n$, are the generalized eigenvalues. $\text{alphan}[j - 1] + \text{alphai}[j - 1]*i$, and $\text{beta}[j - 1]$, $j=1, \dots, n$ are the diagonals of the complex Schur form (<i>S</i> , <i>T</i>) that would result if the 2-by-2 diagonal blocks of the real Schur form of (<i>a</i> , <i>b</i>) were further reduced to triangular form using 2-by-2 complex unitary transformations. If $\text{alphai}[j - 1]$ is zero, then the <i>j</i> -th eigenvalue is real; if positive, then the <i>j</i> -th and (<i>j</i> +1)-st eigenvalues are a complex conjugate pair, with $\text{alphai}[j]$ negative.

Note: the quotients $\alpha[j-1]/\beta[j-1]$ and $\alpha_{\text{hai}}[j-1]/\beta[j-1]$ can easily over- or underflow, and $\beta[j-1]$ might even be zero. Thus, you should avoid computing the ratio α/β by simply dividing α by β . However, α and α_{hai} is always less than and usually comparable with $\text{norm}(a)$ in magnitude, and β is always less than and usually comparable with $\text{norm}(b)$.

For complex flavors:

On exit, $\alpha[j-1][j-1]/\beta[j-1]$, $j=1,\dots,n$, are the generalized eigenvalues. $\alpha[j-1]$, $j=1,\dots,n$ and $\beta[j-1]$, $j=1,\dots,n$ are the diagonals of the complex Schur form (a,b) output by ?gges3. The $\beta[j-1]$ is non-negative real.

Note: the quotient $\alpha[j-1]/\beta[j-1]$ can easily over- or underflow, and $\beta[j-1]$ might even be zero. Thus, you should avoid computing the ratio α/β by simply dividing α by β . However, α is always less than and usually comparable with $\text{norm}(a)$ in magnitude, and β is always less than and usually comparable with $\text{norm}(b)$.

vsl

Array, size ($ldvsl*n$).

If *jobvsl* = 'V', *vsl* contains the left Schur vectors. Not referenced if *jobvsl* = 'N'.

vsr

Array, size ($ldvsr*n$).

If *jobvsr* = 'V', *vsr* contains the right Schur vectors. Not referenced if *jobvsr* = 'N'.

Return Values

This function returns a value *info*.

= 0: successful exit < 0: if *info* = -*i*, the *i*-th argument had an illegal value.

=1,...,n:

for real flavors:

The QZ iteration failed. (a,b) are not in Schur form, but $\alpha[j]$, $\alpha_{\text{hai}}[j]$ and $\beta[j]$ should be correct for $j=\text{info},\dots,n-1$.

The QZ iteration failed. (a,b) are not in Schur form, but $\alpha[j]$ and $\beta[j]$ should be correct for $j=\text{info},\dots,n-1$.

for complex flavors:

> n:

=n+1: other than QZ iteration failed in ?hgeqz.

=n+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Generalized Schur form no longer satisfy $\text{selctg} \neq 0$ This could also be caused due to scaling.

=n+3: reordering failed in ?tgsen.

?ggev

Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.

Syntax

```
lapack_int LAPACKE_sggev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
float* a, lapack_int lda, float* b, lapack_int ldb, float* alphas, float* alphas, float*
beta, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr );

lapack_int LAPACKE_dggeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
double* a, lapack_int lda, double* b, lapack_int ldb, double* alphas, double* alphas,
double* beta, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr );

lapack_int LAPACKE_cggev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* alpha, lapack_complex_float* beta, lapack_complex_float* vl,
lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr );

lapack_int LAPACKE_zggeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* alpha, lapack_complex_double* beta, lapack_complex_double* vl,
lapack_int ldvl, lapack_complex_double* vr, lapack_int ldvr );
```

Include Files

- mkl.h

Description

The ?ggev routine computes the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors for a pair of n -by- n real/complex nonsymmetric matrices (A,B) .

A generalized eigenvalue for a pair of matrices (A,B) is a scalar λ or a ratio $\alpha / \beta = \lambda$, such that $A - \lambda*B$ is singular. It is usually represented as the pair (α, β) , as there is a reasonable interpretation for $\beta = 0$ and even for both being zero.

The right generalized eigenvector $v(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies $A*v(j) = \lambda(j)*B*v(j)$.

The left generalized eigenvector $u(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies $u(j)^H*A = \lambda(j)*u(j)^H*B$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$.

The ?ggev routine replaces the deprecated ?gegv routine.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobvl</i>	Must be 'N' or 'V'. If <i>jobvl</i> = 'N', the left generalized eigenvectors are not computed; If <i>jobvl</i> = 'V', the left generalized eigenvectors are computed.

<i>jobvr</i>	Must be 'N' or 'V'. If <i>jobvr</i> = 'N', the right generalized eigenvectors are not computed; If <i>jobvr</i> = 'V', the right generalized eigenvectors are computed.
<i>n</i>	The order of the matrices <i>A</i> , <i>B</i> , <i>vl</i> , and <i>vr</i> ($n \geq 0$).
<i>a</i> , <i>b</i>	Arrays: <i>a</i> (size at least $\max(1, lda*n)$) is an array containing the <i>n</i> -by- <i>n</i> matrix <i>A</i> (first of the pair of matrices). <i>b</i> (size at least $\max(1, ldb*n)$) is an array containing the <i>n</i> -by- <i>n</i> matrix <i>B</i> (second of the pair of matrices).
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>ldb</i>	The leading dimension of the array <i>b</i> . Must be at least $\max(1, n)$.
<i>ldvl</i> , <i>ldvr</i>	The leading dimensions of the output matrices <i>vl</i> and <i>vr</i> , respectively. Constraints: <i>ldvl</i> ≥ 1 . If <i>jobvl</i> = 'V', <i>ldvl</i> $\geq \max(1, n)$. <i>ldvr</i> ≥ 1 . If <i>jobvr</i> = 'V', <i>ldvr</i> $\geq \max(1, n)$.

Output Parameters

<i>a</i> , <i>b</i>	On exit, these arrays have been overwritten.
<i>alphar</i> , <i>alphai</i>	Arrays, size at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors. See <i>beta</i> .
<i>alpha</i>	Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See <i>beta</i> .
<i>beta</i>	Array, size at least $\max(1, n)$. For real flavors: On exit, $(\text{alphar}[j] + \text{alphai}[j]*i)/\text{beta}[j]$, $j=0, \dots, n-1$, are the generalized eigenvalues. If <i>alphai</i> [<i>j</i>] is zero, then the <i>j</i> -th eigenvalue is real; if positive, then the <i>j</i> -th and (<i>j</i> +1)-st eigenvalues are a complex conjugate pair, with <i>alphai</i> [<i>j</i> +1] negative. For complex flavors: On exit, $\text{alpha}[j]/\text{beta}[j]$, $j=0, \dots, n-1$, are the generalized eigenvalues. See also <i>Application Notes</i> below.
<i>vl</i> , <i>vr</i>	Arrays: <i>vl</i> (size at least $\max(1, ldvl*n)$). Contains the matrix of left generalized eigenvectors <i>VL</i> .

If $jobvl = 'V'$, the left generalized eigenvectors u_j are stored one after another in the columns of VL , in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $abs(Re) + abs(Im) = 1$.

If $jobvl = 'N'$, VL is not referenced.

For real flavors:

If the j -th eigenvalue is real, then the k -th component of the j -th left eigenvector u_j is stored in $vl[(k - 1) + (j - 1)*ldvl]$ for column major layout and in $vl[(k - 1)*ldvl + (j - 1)]$ for row major layout..

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then for $i = \sqrt{-1}$, the k -th components of the j -th left eigenvector u_j are $vl[(k - 1) + (j - 1)*ldvl] + i*vl[(k - 1) + j*ldvl]$ for column major layout and $vl[(k - 1)*ldvl + (j - 1)] + i*vl[(k - 1)*ldvl + j]$ for row major layout. Similarly, the k -th components of left eigenvector $j+1$ u_{j+1} are $vl[(k - 1) + (j - 1)*ldvl] - i*vl[(k - 1) + j*ldvl]$ for column major layout and $vl[(k - 1)*ldvl + (j - 1)] - i*vl[(k - 1)*ldvl + j]$ for row major layout..

For complex flavors:

The k -th component of the j -th left eigenvector u_j is stored in $vl[(k - 1) + (j - 1)*ldvl]$ for column major layout and in $vl[(k - 1)*ldvl + (j - 1)]$ for row major layout.

vr (size at least $\max(1, ldvr*n)$). Contains the matrix of right generalized eigenvectors VR .

If $jobvvr = 'V'$, the right generalized eigenvectors v_j are stored one after another in the columns of VR , in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $abs(Re) + abs(Im) = 1$.

If $jobvvr = 'N'$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then The k -th component of the j -th right eigenvector v_j is stored in $vr[(k - 1) + (j - 1)*ldvr]$ for column major layout and in $vr[(k - 1)*ldvr + (j - 1)]$ for row major layout..

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then the k -th components of the j -th right eigenvector v_j can be computed as $vr[(k - 1) + (j - 1)*ldvr] + i*vr[(k - 1) + j*ldvr]$ for column major layout and $vr[(k - 1)*ldvr + (j - 1)] + i*vr[(k - 1)*ldvr + j]$ for row major layout. Similarly, the k -th components of the right eigenvector $j+1$ v_{j+1} can be computed as $vr[(k - 1) + (j - 1)*ldvr] - i*vr[(k - 1) + j*ldvr]$ for column major layout and $vr[(k - 1)*ldvr + (j - 1)] - i*vr[(k - 1)*ldvr + j]$ for row major layout..

For complex flavors:

The k -th component of the j -th right eigenvector v_j is stored in $vr[(k - 1) + (j - 1)*ldvr]$ for column major layout and in $vr[(k - 1)*ldvr + (j - 1)]$ for row major layout.

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and

i ≤ *n*: the QZ iteration failed. No eigenvectors have been calculated, but *alphas*[*j*], *alphai*[*j*] (for real flavors), or *alpha*[*j*] (for complex flavors), and *beta*[*j*], *j*=*info*, ..., *n* - 1 should be correct.

i > *n*: errors that usually indicate LAPACK problems:

i = *n*+1: other than QZ iteration failed in [hgeqz](#);

i = *n*+2: error return from [tgevc](#).

Application Notes

The quotients *alphas*[*j*]/*beta*[*j*] and *alphai*[*j*]/*beta*[*j*] may easily over- or underflow, and *beta*[*j*] may even be zero. Thus, you should avoid simply computing the ratio. However, *alphas* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

?ggevx

Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

Syntax

```
lapack_int LAPACKE_sggev( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float* alphas,
float* alphai, float* beta, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr,
lapack_int* ilo, lapack_int* ihi, float* lscale, float* rscale, float* abnrm, float*
bbnrm, float* rconde, float* rcondv );
```

```
lapack_int LAPACKE_dggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double*
alphas, double* alphai, double* beta, double* vl, lapack_int ldvl, double* vr,
lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, double* lscale, double* rscale,
double* abnrm, double* bbnrm, double* rconde, double* rcondv );
```

```
lapack_int LAPACKE_cggev( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b,
lapack_int ldb, lapack_complex_float* alpha, lapack_complex_float* beta,
lapack_complex_float* vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr,
lapack_int* ilo, lapack_int* ihi, float* lscale, float* rscale, float* abnrm, float*
bbnrm, float* rconde, float* rcondv );
```

```
lapack_int LAPACKE_zggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double*
b, lapack_int ldb, lapack_complex_double* alpha, lapack_complex_double* beta,
lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int ldvr,
lapack_int* ilo, lapack_int* ihi, double* lscale, double* rscale, double* abnrm,
double* bbnrm, double* rconde, double* rcondv );
```

Include Files

- `mk1.h`

Description

The routine computes for a pair of n -by- n real/complex nonsymmetric matrices (A,B) , the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *lscale*, *rscale*, *abnrm*, and *bbnrm*), reciprocal condition numbers for the eigenvalues (*rconde*), and reciprocal condition numbers for the right eigenvectors (*rcondv*).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar λ or a ratio $\alpha / \beta = \lambda$, such that $A - \lambda*B$ is singular. It is usually represented as the pair (α, β) , as there is a reasonable interpretation for $\beta=0$ and even for both being zero. The right generalized eigenvector $v(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies

$$A*v(j) = \lambda(j)*B*v(j).$$

The left generalized eigenvector $u(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies

$$u(j)^H*A = \lambda(j)*u(j)^H*B$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>balanc</i>	<p>Must be 'N', 'P', 'S', or 'B'. Specifies the balance option to be performed.</p> <p>If <i>balanc</i> = 'N', do not diagonally scale or permute;</p> <p>If <i>balanc</i> = 'P', permute only;</p> <p>If <i>balanc</i> = 'S', scale only;</p> <p>If <i>balanc</i> = 'B', both permute and scale.</p> <p>Computed reciprocal condition numbers will be for the matrices after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.</p>
<i>jobvl</i>	<p>Must be 'N' or 'V'.</p> <p>If <i>jobvl</i> = 'N', the left generalized eigenvectors are not computed;</p> <p>If <i>jobvl</i> = 'V', the left generalized eigenvectors are computed.</p>
<i>jobvr</i>	<p>Must be 'N' or 'V'.</p> <p>If <i>jobvr</i> = 'N', the right generalized eigenvectors are not computed;</p> <p>If <i>jobvr</i> = 'V', the right generalized eigenvectors are computed.</p>
<i>sense</i>	<p>Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.</p> <p>If <i>sense</i> = 'N', none are computed;</p>

	If <i>sense</i> = 'E', computed for eigenvalues only;
	If <i>sense</i> = 'V', computed for eigenvectors only;
	If <i>sense</i> = 'B', computed for eigenvalues and eigenvectors.
<i>n</i>	The order of the matrices <i>A</i> , <i>B</i> , <i>vl</i> , and <i>vr</i> ($n \geq 0$).
<i>a</i> , <i>b</i>	Arrays: <i>a</i> (size at least $\max(1, lda*n)$) is an array containing the <i>n</i> -by- <i>n</i> matrix <i>A</i> (first of the pair of matrices). <i>b</i> (size at least $\max(1, ldb*n)$) is an array containing the <i>n</i> -by- <i>n</i> matrix <i>B</i> (second of the pair of matrices).
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>ldb</i>	The leading dimension of the array <i>b</i> . Must be at least $\max(1, n)$.
<i>ldvl</i> , <i>ldvr</i>	The leading dimensions of the output matrices <i>vl</i> and <i>vr</i> , respectively. Constraints: <i>ldvl</i> ≥ 1 . If <i>jobvl</i> = 'V', <i>ldvl</i> $\geq \max(1, n)$. <i>ldvr</i> ≥ 1 . If <i>jobvr</i> = 'V', <i>ldvr</i> $\geq \max(1, n)$.

Output Parameters

<i>a</i> , <i>b</i>	On exit, these arrays have been overwritten. If <i>jobvl</i> = 'V' or <i>jobvr</i> = 'V' or both, then <i>a</i> contains the first part of the real Schur form of the "balanced" versions of the input <i>A</i> and <i>B</i> , and <i>b</i> contains its second part.
<i>alphas</i> , <i>alphai</i>	Arrays, size at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors. See <i>beta</i> .
<i>alpha</i>	Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See <i>beta</i> .
<i>beta</i>	Array, size at least $\max(1, n)$. For real flavors: On exit, $(\text{alphas}[j] + \text{alphai}[j]*i)/\text{beta}[j]$, $j=0, \dots, n-1$, will be the generalized eigenvalues. If <i>alphai</i> [<i>j</i>] is zero, then the <i>j</i> -th eigenvalue is real; if positive, then the <i>j</i> -th and (<i>j</i> +1)-st eigenvalues are a complex conjugate pair, with <i>alphai</i> [<i>j</i> +1] negative. For complex flavors: On exit, $\text{alpha}[j]/\text{beta}[j]$, $j=0, \dots, n-1$, will be the generalized eigenvalues. See also <i>Application Notes</i> below.
<i>vl</i> , <i>vr</i>	Arrays:

vl (size at least $\max(1, ldvl*n)$).

If $jobvl = 'V'$, the left generalized eigenvectors $u(j)$ are stored one after another in the columns of vl , in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.

If $jobvl = 'N'$, vl is not referenced.

For real flavors:

If the j -th eigenvalue is real, then k -th component of j -th left eigenvector u_j is stored in $vl[(k-1) + (j-1)*ldvl]$ for column major layout and in $vl[(k-1)*ldvl + (j-1)]$ for row major layout..

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then for $i = \text{sqrt}(-1)$, the k -th components of the j -th left eigenvector u_j can be computed as $vl[(k-1) + (j-1)*ldvl] + i*vl[(k-1) + j*ldvl]$ for column major layout and $vl[(k-1)*ldvl + (j-1)] + i*vl[(k-1)*ldvl + j]$ for row major layout. Similarly, the k -th components of the left eigenvector $j+1$ u_{j+1} can be computed as $vl[(k-1) + (j-1)*ldvl] - i*vl[(k-1) + j*ldvl]$ for column major layout and $vl[(k-1)*ldvl + (j-1)] - i*vl[(k-1)*ldvl + j]$ for row major layout..

For complex flavors:

The k -th component of the j -th left eigenvector u_j is stored in $vl[(k-1) + (j-1)*ldvl]$ for column major layout and in $vl[(k-1)*ldvl + (j-1)]$ for row major layout.

vr (size at least $\max(1, ldvr*n)$).

If $jobvvr = 'V'$, the right generalized eigenvectors $v(j)$ are stored one after another in the columns of vr , in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.

If $jobvvr = 'N'$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then the k -th component of the j -th right eigenvector v_j is stored in $vr[(k-1) + (j-1)*ldvr]$ for column major layout and in $vr[(k-1)*ldvr + (j-1)]$ for row major layout..

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then The k -th components of the j -th right eigenvector v_j can be computed as $vr[(k-1) + (j-1)*ldvr] + i*vr[(k-1) + j*ldvr]$ for column major layout and $vr[(k-1)*ldvr + (j-1)] + i*vr[(k-1)*ldvr + j]$ for row major layout. Respectively, the k -th components of right eigenvector $j+1$ v_{j+1} can be computed as $vr[(k-1) + (j-1)*ldvr] - i*vr[(k-1) + j*ldvr]$ for column major layout and $vr[(k-1)*ldvr + (j-1)] - i*vr[(k-1)*ldvr + j]$ for row major layout..

For complex flavors:

The k -th component of the j -th right eigenvector v_j is stored in $vr[(k-1) + (j-1)*ldvr]$ for column major layout and in $vr[(k-1)*ldvr + (j-1)]$ for row major layout.

<i>ilo, ihi</i>	<p><i>ilo</i> and <i>ihi</i> are integer values such that on exit $A_{i,j} = 0$ and $B_{i,j} = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.</p> <p>If <i>balanc</i> = 'N' or 'S', <i>ilo</i> = 1 and <i>ihi</i> = <i>n</i>.</p>
<i>lscale, rscale</i>	<p>Arrays, size at least $\max(1, n)$ each.</p> <p><i>lscale</i> contains details of the permutations and scaling factors applied to the left side of <i>A</i> and <i>B</i>.</p> <p>If $PL(j)$ is the index of the row interchanged with row <i>j</i>, and $DL(j)$ is the scaling factor applied to row <i>j</i>, then</p> <p>$lscale[j - 1] = PL(j)$, for $j = 1, \dots, ilo-1$</p> <p>$= DL(j)$, for $j = ilo, \dots, ihi$</p> <p>$= PL(j)$ for $j = ihi+1, \dots, n$.</p> <p>The order in which the interchanges are made is <i>n</i> to <i>ihi</i>+1, then 1 to <i>ilo</i>-1.</p> <p><i>rscale</i> contains details of the permutations and scaling factors applied to the right side of <i>A</i> and <i>B</i>.</p> <p>If $PR(j)$ is the index of the column interchanged with column <i>j</i>, and $DR(j)$ is the scaling factor applied to column <i>j</i>, then</p> <p>$rscale[j - 1] = PR(j)$, for $j = 1, \dots, ilo-1$</p> <p>$= DR(j)$, for $j = ilo, \dots, ihi$</p> <p>$= PR(j)$ for $j = ihi+1, \dots, n$.</p> <p>The order in which the interchanges are made is <i>n</i> to <i>ihi</i>+1, then 1 to <i>ilo</i>-1.</p>
<i>abnrm, bbnrm</i>	The one-norms of the balanced matrices <i>A</i> and <i>B</i> , respectively.
<i>rconde, rcondv</i>	<p>Arrays, size at least $\max(1, n)$ each.</p> <p>If <i>sense</i> = 'E', or 'B', <i>rconde</i> contains the reciprocal condition numbers of the eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of <i>rconde</i> are set to the same value. Thus <i>rconde</i>[<i>j</i>], <i>rcondv</i>[<i>j</i>], and the <i>j</i>-th columns of <i>vl</i> and <i>vr</i> all correspond to the same eigenpair (but not in general the <i>j</i>-th eigenpair, unless all eigenpairs are selected).</p> <p>If <i>sense</i> = 'N', or 'V', <i>rconde</i> is not referenced.</p> <p>If <i>sense</i> = 'V', or 'B', <i>rcondv</i> contains the estimated reciprocal condition numbers of the eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of <i>rcondv</i> are set to the same value.</p> <p>If the eigenvalues cannot be reordered to compute , <i>rcondv</i>[<i>j</i>] is set to 0; this can only occur when the true value would be very small anyway.</p> <p>If <i>sense</i> = 'N', or 'E', <i>rcondv</i> is not referenced.</p>

Return Values

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and

i ≤ *n*: the QZ iteration failed. No eigenvectors have been calculated, but *alphan*[*j*], *alphai*[*j*] (for real flavors), or *alpha*[*j*] (for complex flavors), and *beta*[*j*], *j*=*info*, ..., *n* - 1 should be correct.

i > *n*: errors that usually indicate LAPACK problems:

i = *n*+1: other than QZ iteration failed in [hgeqz](#);

i = *n*+2: error return from [tgevc](#).

Application Notes

The quotients *alphan*[*j*]/*beta*[*j*] and *alphai*[*j*]/*beta*[*j*] may easily over- or underflow, and *beta*[*j*] may even be zero. Thus, you should avoid simply computing the ratio. However, *alphan* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with norm(*A*) in magnitude, and *beta* always less than and usually comparable with norm(*B*).

?ggev3

Computes the generalized eigenvalues and the left and right generalized eigenvectors for a pair of matrices.

Syntax

```
lapack_int LAPACKESggeev3 (int matrix_layout, char jobvl, char jobvr, lapack_int n,
float * a, lapack_int lda, float * b, lapack_int ldb, float * alphan, float * alphai,
float * beta, float * vl, lapack_int ldvl, float * vr, lapack_int ldvr);
```

```
lapack_int LAPACKEDggeev3 (int matrix_layout, char jobvl, char jobvr, lapack_int n,
double * a, lapack_int lda, double * b, lapack_int ldb, double * alphan, double *
alphai, double * beta, double * vl, lapack_int ldvl, double * vr, lapack_int ldvr);
```

```
lapack_int LAPACKECggeev3 (int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb,
lapack_complex_float * alpha, lapack_complex_float * beta, lapack_complex_float * vl,
lapack_int ldvl, lapack_complex_float * vr, lapack_int ldvr);
```

```
lapack_int LAPACKEZggeev3 (int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb,
lapack_complex_double * alpha, lapack_complex_double * beta, lapack_complex_double *
vl, lapack_int ldvl, lapack_complex_double * vr, lapack_int ldvr);
```

Include Files

- `mkl.h`

Description

For a pair of *n*-by-*n* real or complex nonsymmetric matrices (*A*, *B*), ?ggeev3 computes the generalized eigenvalues, and optionally, the left and right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices (*A*, *B*) is a scalar λ or a ratio $\alpha/\beta = \lambda$, such that $A - \lambda B$ is singular. It is usually represented as the pair (*alpha*, *beta*), as there is a reasonable interpretation for $\beta=0$, and even for both being zero.

For real flavors:

The right eigenvector v_j corresponding to the eigenvalue λ_j of (*A*, *B*) satisfies

$$A * v_j = \lambda_j * B * v_j.$$

The left eigenvector u_j corresponding to the eigenvalue λ_j of (*A*, *B*) satisfies

$$u_j^H * A = \lambda_j * u_j^H * B$$

where u_j^H is the conjugate-transpose of u_j .

For complex flavors:

The right generalized eigenvector v_j corresponding to the generalized eigenvalue λ_j of (A, B) satisfies

$$A * v_j = \lambda_j * B * v_j.$$

The left generalized eigenvector u_j corresponding to the generalized eigenvalues λ_j of (A, B) satisfies

$$u_j^H * A = \lambda_j * u_j^H * B$$

where u_j^H is the conjugate-transpose of u_j .

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>jobvl</i>	= 'N': do not compute the left generalized eigenvectors; = 'V': compute the left generalized eigenvectors.
<i>jobvr</i>	= 'N': do not compute the right generalized eigenvectors; = 'V': compute the right generalized eigenvectors.
<i>n</i>	The order of the matrices A , B , VL , and VR . $n \geq 0$.
<i>a</i>	Array, size ($lda * n$). On entry, the matrix A in the pair (A, B) .
<i>lda</i>	The leading dimension of a . $lda \geq \max(1, n)$.
<i>b</i>	Array, size ($ldb * n$). On entry, the matrix B in the pair (A, B) .
<i>ldb</i>	The leading dimension of b . $ldb \geq \max(1, n)$.
<i>ldvl</i>	The leading dimension of the matrix VL . $ldvl \geq 1$, and if <i>jobvl</i> = 'V', $ldvl \geq n$.
<i>ldvr</i>	The leading dimension of the matrix VR . $ldvr \geq 1$, and if <i>jobvr</i> = 'V', $ldvr \geq n$.

Output Parameters

<i>a</i>	On exit, a is overwritten.
<i>b</i>	On exit, b is overwritten.
<i>alphar</i>	Array, size (n).
<i>alpha</i>	Array, size (n).

<i>alpha</i>	Array, size (<i>n</i>).
<i>beta</i>	<p>Array, size (<i>n</i>).</p> <p>For real flavors:</p> <p>On exit, $(\text{alphar}[j] + \text{alphai}[j]*i)/\text{beta}[j]$, $j=0,\dots,n-1$, are the generalized eigenvalues. If $\text{alphai}[j-1]$ is zero, then the j-th eigenvalue is real; if positive, then the j-th and $(j+1)$-st eigenvalues are a complex conjugate pair, with $\text{alphai}[j]$ negative.</p> <p>Note: the quotients $\text{alphar}[j-1]/\text{beta}[j-1]$ and $\text{alphai}[j-1]/\text{beta}[j-1]$ can easily over- or underflow, and $\text{beta}(j)$ might even be zero. Thus, you should avoid computing the ratio alpha/beta by simply dividing alpha by beta. However, alphar and alphai are always less than and usually comparable with $\text{norm}(A)$ in magnitude, and beta is always less than and usually comparable with $\text{norm}(B)$.</p> <p>For complex flavors:</p> <p>On exit, $\text{alpha}[j]/\text{beta}[j]$, $j=0,\dots,n-1$, are the generalized eigenvalues.</p> <p>Note: the quotients $\text{alpha}[j-1]/\text{beta}[j-1]$ may easily over- or underflow, and $\text{beta}(j)$ can even be zero. Thus, you should avoid computing the ratio alpha/beta by simply dividing alpha by beta. However, alpha is always less than and usually comparable with $\text{norm}(A)$ in magnitude, and beta is always less than and usually comparable with $\text{norm}(B)$.</p>
<i>v1</i>	<p>Array, size ($\text{ldv1} * n$).</p> <p>For real flavors:</p> <p>If $\text{jobv1} = 'V'$, the left eigenvectors u_j are stored one after another in the columns of <i>v1</i>, in the same order as their eigenvalues. If the j-th eigenvalue is real, then u_j = the j-th column of <i>v1</i>. If the j-th and $(j+1)$-st eigenvalues form a complex conjugate pair, then the real part of u_j = the j-th column of <i>v1</i> and the imaginary part of v_j = the $(j+1)$-st column of <i>v1</i>.</p> <p>Each eigenvector is scaled so the largest component has $\text{abs}(\text{real part}) + \text{abs}(\text{imag. part}) = 1$.</p> <p>Not referenced if $\text{jobv1} = 'N'$.</p> <p>For complex flavors:</p> <p>If $\text{jobv1} = 'V'$, the left generalized eigenvectors u_j are stored one after another in the columns of <i>v1</i>, in the same order as their eigenvalues.</p> <p>Each eigenvector is scaled so the largest component has $\text{abs}(\text{real part}) + \text{abs}(\text{imag. part}) = 1$.</p> <p>Not referenced if $\text{jobv1} = 'N'$.</p>
<i>vr</i>	<p>Array, size ($\text{ldvr} * n$).</p> <p>For real flavors:</p>

If $jobvr = 'V'$, the right eigenvectors v_j are stored one after another in the columns of vr , in the same order as their eigenvalues. If the j -th eigenvalue is real, then v_j = the j -th column of vr . If the j -th and $(j + 1)$ -st eigenvalues form a complex conjugate pair, then the real part of v_j = the j -th column of vr and the imaginary part of v_j = the $(j + 1)$ -st column of vr .

Each eigenvector is scaled so the largest component has $\text{abs}(\text{real part}) + \text{abs}(\text{imag. part}) = 1$.

Not referenced if $jobvr = 'N'$.

For complex flavors:

If $jobvr = 'V'$, the right generalized eigenvectors v_j are stored one after another in the columns of vr , in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $\text{abs}(\text{real part}) + \text{abs}(\text{imag. part}) = 1$.

Not referenced if $jobvr = 'N'$.

Return Values

This function returns a value *info*.

= 0: successful exit

< 0: if $info = -i$, the i -th argument had an illegal value.

= 1, ..., n :

for real flavors:

The QZ iteration failed. No eigenvectors have been calculated, but $\alpha_{phar}[j]$, $\alpha_{phar}[j]$ and $\beta_{eta}[j]$ should be correct for $j = info, \dots, n - 1$.

for complex flavors:

The QZ iteration failed. No eigenvectors have been calculated, but $\alpha_{pha}[j]$ and $\beta_{eta}[j]$ should be correct for $j = info, \dots, n - 1$.

> n :

= $n + 1$: other than QZ iteration failed in ?hgeqz,

= $n + 2$: error return from ?tgevc.

LAPACK Auxiliary Routines

Routine naming conventions, mathematical notation, and matrix storage schemes used for LAPACK auxiliary routines are the same as for the driver and computational routines described in previous chapters.

?lacgv

Conjugates a complex vector.

Syntax

```
lapack_int LAPACKC_lacgv (lapack_int n, lapack_complex_float* x, lapack_int incx);
```

```
lapack_int LAPACKC_zlacgv (lapack_int n, lapack_complex_double* x, lapack_int incx);
```

Include Files

- `mk1.h`

Description

The routine conjugates a complex vector x of length n and increment $incx$ (see "[Vector Arguments in BLAS](#)" in Appendix B).

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

n	The length of the vector x ($n \geq 0$).
x	Array, dimension $(1 + (n-1) * incx)$. Contains the vector of length n to be conjugated.
$incx$	The spacing between successive elements of x .

Output Parameters

x	On exit, overwritten with <code>conjg(x)</code> .
-----	---

?lacrm

Multiplies a complex matrix by a square real matrix.

Syntax

```
call clacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
call zlacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

Include Files

- `mk1.h`

Description

The routine performs a simple matrix-matrix multiplication of the form

$$C = A * B,$$

where A is m -by- n and complex, B is n -by- n and real, C is m -by- n and complex.

Input Parameters

m	INTEGER. The number of rows of the matrix A and of the matrix C ($m \geq 0$).
n	INTEGER. The number of columns and rows of the matrix B and the number of columns of the matrix C ($n \geq 0$).
a	COMPLEX for <code>clacrm</code> DOUBLE COMPLEX for <code>zlacrm</code>

Array, DIMENSION(lda , n). Contains the m -by- n matrix A .

lda INTEGER. The leading dimension of the array a , $lda \geq \max(1, m)$.

b REAL for `clacrm`
DOUBLE PRECISION for `zlacrm`

Array, DIMENSION(ldb , n). Contains the n -by- n matrix B .

ldb INTEGER. The leading dimension of the array b , $ldb \geq \max(1, n)$.

ldc INTEGER. The leading dimension of the output array c , $ldc \geq \max(1, n)$.

$rwork$ REAL for `clacrm`
DOUBLE PRECISION for `zlacrm`

Workspace array, DIMENSION($2*m*n$).

Output Parameters

c COMPLEX for `clacrm`
DOUBLE COMPLEX for `zlacrm`

Array, DIMENSION(ldc , n). Contains the m -by- n matrix C .

?ssyconv

Converts a symmetric matrix given by a triangular matrix factorization into two matrices and vice versa.

Syntax

```
lapack_int LAPACKE_ssyconv (int matrix_layout, char uplo, char way, lapack_int n, float
* a, lapack_int lda, const lapack_int * ipiv, float * e);

lapack_int LAPACKE_dsyconv (int matrix_layout, char uplo, char way, lapack_int n,
double* a, lapack_int lda, const lapack_int * ipiv, double * e);

lapack_int LAPACKE_csyconv (int matrix_layout, char uplo, char way, lapack_int n,
lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_float
* e);

lapack_int LAPACKE_zsyconv (int matrix_layout, char uplo, char way, lapack_int n,
lapack_complex_double* a, lapack_int lda, const lapack_int * ipiv,
lapack_complex_double * e);
```

Include Files

- `mkl.h`

Description

The routine converts matrix A , which results from a triangular matrix factorization, into matrices L and D and vice versa. The routine returns non-diagonalized elements of D and applies or reverses permutation done with the triangular matrix factorization.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Must be 'U' or 'L'. Indicates whether the details of the factorization are stored as an upper or lower triangular matrix: If <code>uplo = 'U'</code> : the upper triangular, $A = U * D * U^T$. If <code>uplo = 'L'</code> : the lower triangular, $A = L * D * L^T$.
<code>way</code>	Must be 'C' or 'R'.
<code>n</code>	The order of matrix A ; $n \geq 0$.
<code>a</code>	Array of size $\max(1, lda * n)$. The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <code>?sytrf</code> .
<code>lda</code>	The leading dimension of a ; $lda \geq \max(1, n)$.
<code>ipiv</code>	Array, size at least $\max(1, n)$. Details of the interchanges and the block structure of D , as returned by <code>?sytrf</code> .

Output Parameters

<code>e</code>	Array of size $\max(1, n)$ containing the superdiagonal/subdiagonal of the symmetric 1-by-1 or 2-by-2 block diagonal matrix D in $L * D * L^T$.
----------------	--

Return Values

<code>info</code>	If <code>info = 0</code> , the execution is successful. If <code>info < 0</code> , the i -th parameter had an illegal value. If <code>info = -1011</code> , memory allocation error occurred.
-------------------	--

See Also

[?sytrf](#)

?syr

Performs the symmetric rank-1 update of a complex symmetric matrix.

Syntax

```
lapack_int LAPACKE_csyrr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float alpha, const lapack_complex_float * x, lapack_int incx,
lapack_complex_float * a, lapack_int lda);

lapack_int LAPACKE_zsyrr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double alpha, const lapack_complex_double * x, lapack_int incx,
lapack_complex_double * a, lapack_int lda);
```

Include Files

- `mk1.h`

Description

The routine performs the symmetric rank 1 operation defined as

$$a := \alpha * x * x^H + a,$$

where:

- α is a complex scalar.
- x is an n -element complex vector.
- a is an n -by- n complex symmetric matrix.

These routines have their real equivalents in BLAS (see [?syr](#) in Chapter "BLAS and Sparse BLAS Routines").

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Specifies whether the upper or lower triangular part of the array a is used: If <code>uplo</code> = 'U' or 'u', then the upper triangular part of the array a is used. If <code>uplo</code> = 'L' or 'l', then the lower triangular part of the array a is used.
<code>n</code>	Specifies the order of the matrix a . The value of n must be at least zero.
<code>alpha</code>	Specifies the scalar α .
<code>x</code>	Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n -element vector x .
<code>incx</code>	Specifies the increment for the elements of x . The value of <code>incx</code> must not be zero.
<code>a</code>	Array, size $\max(1, \text{lda} * n)$. Before entry with <code>uplo</code> = 'U' or 'u', the leading n -by- n upper triangular part of the array a must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of a is not referenced. Before entry with <code>uplo</code> = 'L' or 'l', the leading n -by- n lower triangular part of the array a must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of a is not referenced.
<code>lda</code>	Specifies the leading dimension of a as declared in the calling (sub)program. The value of <code>lda</code> must be at least $\max(1, n)$.

Output Parameters

<code>a</code>	With <code>uplo</code> = 'U' or 'u', the upper triangular part of the array a is overwritten by the upper triangular part of the updated matrix. With <code>uplo</code> = 'L' or 'l', the lower triangular part of the array a is overwritten by the lower triangular part of the updated matrix.
----------------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

i?max1

Finds the index of the vector element whose real part has maximum absolute value.

Syntax

```
MKL_INT icmax1(const MKL_INT*n, const MKL_Complex8*cx, const MKL_INT*incx)
```

```
MKL_INT izmax1(const MKL_INT*n, const MKL_Complex16*cx, const MKL_INT*incx)
```

Include Files

- mkl.h

Description

Given a complex vector *cx*, the *i?max1* functions return the index of the first vector element of maximum absolute value. These functions are based on the BLAS functions *icamax/izamax*, but using the absolute value of components. They are designed for use with *clacon/zlacon*.

Input Parameters

<i>n</i>	Specifies the number of elements in the vector <i>cx</i> .
<i>cx</i>	Array, size at least $(1 + (n-1) * \text{abs}(incx))$. Contains the input vector.
<i>incx</i>	Specifies the spacing between successive elements of <i>cx</i> .

Return Values

Index of the vector element of maximum absolute value.

?sum1

Forms the 1-norm of the complex vector using the true absolute value.

Syntax

```
float scsum1(const MKL_INT*n, const MKL_Complex8*cx, const MKL_INT*incx)
```

```
double dzsum1(const MKL_INT*n, const MKL_Complex16*cx, const MKL_INT*incx)
```

Include Files

- mkl.h

Description

Given a complex vector *cx*, *scsum1/dzsum1* functions take the sum of the absolute values of vector elements and return a single/double precision result, respectively. These functions are based on *scasum/dzasum* from Level 1 BLAS, but use the true absolute value and were designed for use with *clacon/zlacon*.

Input Parameters

<i>n</i>	Specifies the number of elements in the vector <i>cx</i> .
<i>cx</i>	Array, size at least $(1 + (n-1) * \text{abs}(\text{incx}))$. Contains the input vector whose elements will be summed.
<i>incx</i>	Specifies the spacing between successive elements of <i>cx</i> (<i>incx</i> > 0).

Return Values

Sum of absolute values.

?gelq2

Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
lapack_int LAPACKE_sgelq2 (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, float* tau);

lapack_int LAPACKE_dgelq2 (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, double * tau);

lapack_int LAPACKE_cgelq2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);

lapack_int LAPACKE_zgelq2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine computes an *LQ* factorization of a real/complex *m*-by-*n* matrix *A* as $A = L * Q$.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of $\min(m, n)$ *elementary reflectors* :

$Q = H(k) \dots H(2) H(1)$ (or $Q = H(k)^H \dots H(2)^H H(1)^H$ for complex flavors), where $k = \min(m, n)$

Each *H*(*i*) has the form

$H(i) = I - \tau * v * v^T$ for real flavors, or

$H(i) = I - \tau * v * v^H$ for complex flavors,

where *tau* is a real/complex scalar stored in *tau*(*i*), and *v* is a real/complex vector with $v_{1:i-1} = 0$ and $v_i = 1$.

On exit, the j -th ($i+1 \leq j \leq n$) component of vector v (for real functions) or its conjugate (for complex functions) is stored in $a[i - 1 + lda*(j - 1)]$ for column major layout or in $a[j - 1 + lda*(i - 1)]$ for row major layout.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

m	The number of rows in the matrix A ($m \geq 0$).
n	The number of columns in A ($n \geq 0$).
a	Array, size at least $\max(1, lda*n)$ for column major and $\max(1, lda*m)$ for row major layout. Array a contains the m -by- n matrix A .
lda	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

a	Overwritten by the factorization data as follows: on exit, the elements on and below the diagonal of the array a contain the m -by- $\min(n, m)$ lower trapezoidal matrix L (L is lower triangular if $n \geq m$); the elements above the diagonal, with the array τ , represent the orthogonal/unitary matrix Q as a product of $\min(n, m)$ elementary reflectors.
τ	Array, size at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = - i , the i -th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?geqr2

Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
lapack_int LAPACKE_sgeqr2 (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, float* tau);

lapack_int LAPACKE_dgeqr2 (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, double* tau);

lapack_int LAPACKE_cgeqr2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);

lapack_int LAPACKE_zgeqr2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- `mkl.h`

Description

The routine computes a *QR* factorization of a real/complex *m*-by-*n* matrix *A* as $A = Q \cdot R$.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of $\min(m, n)$ *elementary reflectors* :

$$Q = H(1) * H(2) * \dots * H(k), \text{ where } k = \min(m, n)$$

Each *H*(*i*) has the form

$$H(i) = I - \tau \cdot v \cdot v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau \cdot v \cdot v^H \text{ for complex flavors}$$

where *tau* is a real/complex scalar stored in *tau*[*i*], and *v* is a real/complex vector with $v_{1:i-1} = 0$ and $v_i = 1$.

On exit, $v_{i+1:m}$ is stored in *a*(*i*+1:*m*, *i*).

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>m</i>	The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	The number of columns in <i>A</i> ($n \geq 0$).
<i>a</i>	Array, size at least $\max(1, lda \cdot n)$ for column major and $\max(1, lda \cdot m)$ for row major layout. Array <i>a</i> contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> .
<i>lda</i>	The leading dimension of <i>a</i> ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: on exit, the elements on and above the diagonal of the array <i>a</i> contain the $\min(n, m)$ -by- <i>n</i> upper trapezoidal matrix <i>R</i> (<i>R</i> is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors.
<i>tau</i>	Array, size at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?geqrt2

Computes a QR factorization of a general real or complex matrix using the compact WY representation of Q .

Syntax

```
lapack_int LAPACKGE_sgeqrt2 (int matrix_layout, lapack_int m, lapack_int n, float * a,
lapack_int lda, float * t, lapack_int ldt );

lapack_int LAPACKGE_dgeqrt2 (int matrix_layout, lapack_int m, lapack_int n, double * a,
lapack_int lda, double * t, lapack_int ldt );

lapack_int LAPACKGE_cgeqrt2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float * a, lapack_int lda, lapack_complex_float * t, lapack_int ldt );

lapack_int LAPACKGE_zgeqrt2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double * a, lapack_int lda, lapack_complex_double * t, lapack_int ldt );
```

Include Files

- mkl.h

Description

The strictly lower triangular matrix V contains the elementary reflectors $H(i)$ in the i th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix V is

$$V = \begin{bmatrix} 1 & & & \\ v_1 & 1 & & \\ v_1 & v_2 & 1 & \\ v_1 & v_2 & v_3 & \\ v_1 & v_2 & v_3 & \end{bmatrix}$$

where v_i represents the vector that defines $H(i)$. The vectors are returned in the lower triangular part of array a .

NOTE

The 1s along the diagonal of V are not stored in a .

The block reflector H is then given by

$H = I - V^* T^* V^T$ for real flavors, and

$H = I - V^* T^* V^H$ for complex flavors,

where V^T is the transpose and V^H is the conjugate transpose of V .

Input Parameters

m	The number of rows in the matrix A ($m \geq n$).
n	The number of columns in A ($n \geq 0$).
a	Array, size at least $\max(1, lda*n)$ for column major and $\max(1, lda*m)$ for row major layout. Array a contains the m -by- n matrix A .
lda	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
ldt	The leading dimension of t ; at least $\max(1, n)$.

Output Parameters

a	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the n -by- n upper triangular matrix R . The elements below the diagonal are the columns of V .
t	Array, size at least $\max(1, ldt*n)$. The n -by- n upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector T . The elements below the diagonal are not used.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0 and *info* = -*i*, the *i*th argument had an illegal value.

If *info* = -1011, memory allocation error occurred.

?geqrt3

Recursively computes a QR factorization of a general real or complex matrix using the compact WY representation of Q.

Syntax

```
lapack_int LAPACKE_sgeqrt3 (int matrix_layout , lapack_int m , lapack_int n , float *
a , lapack_int lda , float * t , lapack_int ldt );

lapack_int LAPACKE_dgeqrt3 (int matrix_layout , lapack_int m , lapack_int n , double *
a , lapack_int lda , double * t , lapack_int ldt );

lapack_int LAPACKE_cgeqrt3 (int matrix_layout , lapack_int m , lapack_int n ,
lapack_complex_float * a , lapack_int lda , lapack_complex_float * t , lapack_int
ldt );

lapack_int LAPACKE_zgeqrt3 (int matrix_layout , lapack_int m , lapack_int n ,
lapack_complex_double * a , lapack_int lda , lapack_complex_double * t , lapack_int
ldt );
```

Include Files

- `mk1.h`

Description

The strictly lower triangular matrix V contains the elementary reflectors $H(i)$ in the i th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix V is

$$V = \begin{bmatrix} 1 & & & \\ v_1 & 1 & & \\ v_1 & v_2 & 1 & \\ v_1 & v_2 & v_3 & \\ v_1 & v_2 & v_3 & \end{bmatrix}$$

where v_i represents one of the vectors that define $H(i)$. The vectors are returned in the lower part of triangular array a .

NOTE

The 1s along the diagonal of V are not stored in a .

The block reflector H is then given by

$H = I - V^* T^* V^T$ for real flavors, and

$H = I - V^* T^* V^H$ for complex flavors,

where V^T is the transpose and V^H is the conjugate transpose of V .

Input Parameters

m	The number of rows in the matrix A ($m \geq n$).
n	The number of columns in A ($n \geq 0$).
a	Array, size at least $\max(1, lda * n)$ for column major and $\max(1, lda * m)$ for row major layout. Array a contains the m -by- n matrix A .
lda	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
ldt	The leading dimension of t ; at least $\max(1, n)$.

Output Parameters

a	The elements on and above the diagonal of the array contain the n -by- n upper triangular matrix R . The elements below the diagonal are the columns of V .
t	Array, size ldt by n . The n -by- n upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector T . The elements below the diagonal are not used.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0 and *info* = -*i*, the *i*th argument had an illegal value.

If *info* = -1011, memory allocation error occurred.

?getf2

Computes the LU factorization of a general m -by- n matrix using partial pivoting with row interchanges (unblocked algorithm).

Syntax

```
lapack_int LAPACKESgetf2 (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKEDgetf2 (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, lapack_int * ipiv);
```

```
lapack_int LAPACKE_cgetf2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_zgetf2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_int * ipiv);
```

Include Files

- mkl.h

Description

The routine computes the LU factorization of a general m -by- n matrix A using partial pivoting with row interchanges. The factorization has the form

$$A = P * L * U$$

where p is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

m	The number of rows in the matrix A ($m \geq 0$).
n	The number of columns in A ($n \geq 0$).
a	Array, size at least $\max(1, lda * n)$ for column major and $\max(1, lda * m)$ for row major layout. Array a contains the m -by- n matrix A .
lda	The leading dimension of a ; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

a	Overwritten by L and U . The unit diagonal elements of L are not stored.
$ipiv$	Array, size at least $\max(1, \min(m, n))$. The pivot indices: for $1 \leq i \leq n$, row i was interchanged with row $ipiv(i)$.

Return Values

This function returns a value *info*.

If *info* = $-i$, the i -th parameter had an illegal value.

If *info* = $i > 0$, u_{ij} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

If *info* = -1011, memory allocation error occurred.

?lacn2

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

C:

```
lapack_int LAPACKE_slacn2 (lapack_int n, float * v, float * x, lapack_int * isgn, float
* est, lapack_int * kase, lapack_int * isave);

lapack_int LAPACKE_clacn2 (lapack_int n, lapack_complex_float * v, lapack_complex_float
* x, float * est, lapack_int * kase, lapack_int * isave);

lapack_int LAPACKE_dlacn2 (lapack_int n, double * v, double * x, lapack_int * isgn,
double * est, lapack_int * kase, lapack_int * isave);

lapack_int LAPACKE_zlacn2 (lapack_int n, lapack_complex_double * v,
lapack_complex_double * x, double * est, lapack_int * kase, lapack_int * isave);
```

Include Files

- mkl.h

Description

The routine estimates the 1-norm of a square, real or complex matrix A . Reverse communication is used for evaluating matrix-vector products.

Input Parameters

n	The order of the matrix A ($n \geq 1$).
v, x	Arrays, size (n) each. v is a workspace array. x is used as input after an intermediate return.
$isgn$	Workspace array, size (n), used with real flavors only.
est	On entry with $kase$ set to 1 or 2, and $isave(1) = 1$, est must be unchanged from the previous call to the routine.
$kase$	On the initial call to the routine, $kase$ must be set to 0.
$isave$	Array, size (3). Contains variables from the previous call to the routine.

Output Parameters

est	An estimate (a lower bound) for $\text{norm}(A)$.
$kase$	On an intermediate return, $kase$ is set to 1 or 2, indicating whether x is overwritten by $A*x$ or A^T*x for real flavors and $A*x$ or A^H*x for complex flavors. On the final return, $kase$ is set to 0.
v	On the final return, $v = A*w$, where $est = \text{norm}(v) / \text{norm}(w)$ (w is not returned).
x	On an intermediate return, x is overwritten by $A*x$, if $kase = 1$,

$A^T * x$, if $kase = 2$ (for real flavors),

$A^H * x$, if $kase = 2$ (for complex flavors),

and the routine must be re-called with all the other parameters unchanged.

isave

This parameter is used to save variables between calls to the routine.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
lapack_int LAPACKE_slacpy (int matrix_layout, char uplo, lapack_int m, lapack_int n,
const float* a, lapack_int lda, float* b, lapack_int ldb);
```

```
lapack_int LAPACKE_dlacpy (int matrix_layout, char uplo, lapack_int m, lapack_int n,
const double* a, lapack_int lda, double* b, lapack_int ldb);
```

```
lapack_int LAPACKE_clacpy (int matrix_layout, char uplo, lapack_int m, lapack_int n,
const lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int
ldb);
```

```
lapack_int LAPACKE_zlacpy (int matrix_layout, char uplo, lapack_int m, lapack_int n,
const lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int
ldb);
```

Include Files

- mkl.h

Description

The routine copies all or part of a two-dimensional matrix *A* to another matrix *B*.

Input Parameters

A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

uplo

Specifies the part of the matrix *A* to be copied to *B*.

If *uplo* = 'U', the upper triangular part of *A*;

if *uplo* = 'L', the lower triangular part of *A*.

Otherwise, all of the matrix *A* is copied.

m

The number of rows in the matrix *A* ($m \geq 0$).

n

The number of columns in *A* ($n \geq 0$).

<i>a</i>	Array, size at least $\max(1, lda*n)$ for column major and $\max(1, lda*m)$ for row major layout. <i>A</i> contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> . If <i>uplo</i> = 'U', only the upper triangle or trapezoid is accessed; if <i>uplo</i> = 'L', only the lower triangle or trapezoid is accessed.
<i>lda</i>	The leading dimension of <i>a</i> ; $lda \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.
<i>ldb</i>	The leading dimension of the output array <i>b</i> ; $ldb \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

<i>b</i>	Array, size at least $\max(1, ldb*n)$ for column major and $\max(1, ldb*m)$ for row major layout. Array <i>a</i> contains the <i>m</i> -by- <i>n</i> matrix <i>B</i> . On exit, $B = A$ in the locations specified by <i>uplo</i> .
----------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

?lakf2

Forms a matrix containing Kronecker products between the given matrices.

Syntax

```
void slakf2 (lapack_int *m, lapack_int *n, float *a, lapack_int *lda, float *b, float
*d, float *e, float *z, lapack_int *ldz);

void dlakf2 (lapack_int *m, lapack_int *n, double *a, lapack_int *lda, double *b, double
*d, double *e, double *z, lapack_int *ldz);

void clakf2 (lapack_int *m, lapack_int *n, lapack_complex *a, lapack_int *lda,
lapack_complex *b, lapack_complex *d, lapack_complex *e, lapack_complex *z, lapack_int
*ldz);

void zlakf2 (lapack_int *m, lapack_int *n, lapack_complex_double *a, lapack_int *lda,
lapack_complex_double *b, lapack_complex_double *d, lapack_complex_double *e,
lapack_complex_double *z, lapack_int *ldz);
```

Include Files

- mkl.h

Description

The routine ?lakf2 forms the $2*m*n$ by $2*m*n$ matrix *Z*.

$$Z = \begin{bmatrix} \text{kron}(In, A) & -\text{kron}(B^T, Im) \\ \text{kron}(In, D) & -\text{kron}(E^T, Im) \end{bmatrix}$$

where I_n is the identity matrix of size n and X^T is the transpose of X . $\text{kron}(X, Y)$ is the Kronecker product between the matrices X and Y .

Input Parameters

m	Size of matrix, $m \geq 1$
n	Size of matrix, $n \geq 1$
a	Array, size lda -by- n . The matrix A in the output matrix Z .
lda	The leading dimension of a , b , d , and e . $lda \geq m+n$.
b	Array, size lda by n . Matrix used in forming the output matrix Z .
d	Array, size lda by m . Matrix used in forming the output matrix Z .
e	Array, size lda by n . Matrix used in forming the output matrix Z .
ldz	The leading dimension of Z . $ldz \geq 2 * m * n$.

Output Parameters

z	Array, size ldz -by- $2 * m * n$. The resultant Kronecker $m * n * 2$ -by- $m * n * 2$ matrix.
-----	---

?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.

Syntax

```
float LAPACKE_slange (int matrix_layout, char norm, lapack_int m, lapack_int n, const
float * a, lapack_int lda);

double LAPACKE_dlange (int matrix_layout, char norm, lapack_int m, lapack_int n, const
double * a, lapack_int lda);

float LAPACKE_clange (int matrix_layout, char norm, lapack_int m, lapack_int n, const
lapack_complex_float * a, lapack_int lda);

double LAPACKE_zlange (int matrix_layout, char norm, lapack_int m, lapack_int n, const
lapack_complex_double * a, lapack_int lda);
```

Include Files

- mkl.h

Description

The function ?lange returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex matrix A .

Input Parameters

A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>norm</i>	<p>Specifies the value to be returned by the routine:</p> <p>= 'M' or 'm': $val = \max(\text{abs}(A_{ij})),$ largest absolute value of the matrix <i>A</i>.</p> <p>= '1' or 'O' or 'o': $val = \text{norm1}(A),$ 1-norm of the matrix <i>A</i> (maximum column sum),</p> <p>= 'I' or 'i': $val = \text{normI}(A),$ infinity norm of the matrix <i>A</i> (maximum row sum),</p> <p>= 'F', 'f', 'E' or 'e': $val = \text{normF}(A),$ Frobenius norm of the matrix <i>A</i> (square root of sum of squares).</p>
<i>m</i>	<p>The number of rows of the matrix <i>A</i>.</p> <p>$m \geq 0$. When $m = 0$, <i>?lange</i> is set to zero.</p>
<i>n</i>	<p>The number of columns of the matrix <i>A</i>.</p> <p>$n \geq 0$. When $n = 0$, <i>?lange</i> is set to zero.</p>
<i>a</i>	<p>Array, size at least $\max(1, lda \cdot n)$ for column major and $\max(1, lda \cdot m)$ for row major layout. Array <i>a</i> contains the <i>m</i>-by-<i>n</i> matrix <i>A</i>.</p>
<i>lda</i>	<p>The leading dimension of the array <i>a</i>.</p> <p>$lda \geq \max(n, 1)$ for column major layout and $\max(1, n)$ for row major layout.</p>

?lansy

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.

Syntax

```
float LAPACKE_slansy (int matrix_layout, char norm, char uplo, lapack_int n, const
float * a, lapack_int lda);

double LAPACKE_dlansy (int matrix_layout, char norm, char uplo, lapack_int n, const
double * a, lapack_int lda);

float LAPACKE_clansy (int matrix_layout, char norm, char uplo, lapack_int n, const
lapack_complex_float * a, lapack_int lda);

double LAPACKE_zlansy (int matrix_layout, char norm, char uplo, lapack_int n, const
lapack_complex_double * a, lapack_int lda);
```

Include Files

- mkl.h

Description

The function *?lansy* returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix *A*.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<code>norm</code>	<p>Specifies the value to be returned by the routine:</p> <ul style="list-style-type: none"> = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<code>uplo</code>	<p>Specifies whether the upper or lower triangular part of the symmetric matrix A is to be referenced.</p> <ul style="list-style-type: none"> = 'U': Upper triangular part of A is referenced. = 'L': Lower triangular part of A is referenced
<code>n</code>	The order of the matrix A . $n \geq 0$. When $n = 0$, <code>?lanzy</code> is set to zero.
<code>a</code>	<p>Array, size at least $\max(1, lda * n)$. The symmetric matrix A.</p> <p>If <code>uplo = 'U'</code>, the leading n-by-n upper triangular part of a contains the upper triangular part of the matrix A, and the strictly lower triangular part of a is not referenced.</p> <p>If <code>uplo = 'L'</code>, the leading n-by-n lower triangular part of a contains the lower triangular part of the matrix A, and the strictly upper triangular part of a is not referenced.</p>
<code>lda</code>	<p>The leading dimension of the array a.</p> <p>$lda \geq \max(n, 1)$.</p>

?lanhe

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.

Syntax

```
float LAPACKE_clanhe (int matrix_layout, char norm, char uplo, lapack_int n, const
lapack_complex_float * a, lapack_int lda);

double LAPACKE_zlanhe (int matrix_layout, char norm, char uplo, lapack_int n, const
lapack_complex_double * a, lapack_int lda);
```

Include Files

- `mkl.h`

Description

The function `?lanhe` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix A .

Input Parameters

A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>norm</i>	Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A . = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	Specifies whether the upper or lower triangular part of the Hermitian matrix A is to be referenced. = 'U': Upper triangular part of A is referenced. = 'L': Lower triangular part of A is referenced
<i>n</i>	The order of the matrix A . $n \geq 0$. When $n = 0$, <code>?lanhe</code> is set to zero.
<i>a</i>	Array, size at least $\max(1, lda \cdot n)$. The Hermitian matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.
<i>lda</i>	The leading dimension of the array a . $lda \geq \max(n, 1)$.

?lantr

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.

Syntax

```
float LAPACKE_slantr (char * norm, char * uplo, char * diag, lapack_int * m, lapack_int * n, const float * a, lapack_int * lda, float * work);

double LAPACKE_dlantr (char * norm, char * uplo, char * diag, lapack_int * m, lapack_int * n, const double * a, lapack_int * lda, double * work);

float LAPACKE_clantr (char * norm, char * uplo, char * diag, lapack_int * m, lapack_int * n, const lapack_complex_float * a, lapack_int * lda, float * work);

double LAPACKE_zlantr (char * norm, char * uplo, char * diag, lapack_int * m, lapack_int * n, const lapack_complex_double * a, lapack_int * lda, double * work);
```

Include Files

- `mkl.h`

Description

The function `?lantr` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix A .

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>norm</i>	<p>Specifies the value to be returned by the routine:</p> <ul style="list-style-type: none"> = 'M' or 'm': $val = \max(\text{abs}(A_{ij})),$ largest absolute value of the matrix A. = '1' or 'O' or 'o': $val = \text{norm1}(A),$ 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A),$ infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A),$ Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	<p>Specifies whether the matrix A is upper or lower trapezoidal.</p> <ul style="list-style-type: none"> = 'U': Upper trapezoidal = 'L': Lower trapezoidal. <p>Note that A is triangular instead of trapezoidal if $m = n$.</p>
<i>diag</i>	<p>Specifies whether or not the matrix A has unit diagonal.</p> <ul style="list-style-type: none"> = 'N': Non-unit diagonal = 'U': Unit diagonal.
<i>m</i>	<p>The number of rows of the matrix A. $m \geq 0$, and if <i>uplo</i> = 'U', $m \leq n$.</p> <p>When $m = 0$, <code>?lantr</code> is set to zero.</p>
<i>n</i>	<p>The number of columns of the matrix A. $n \geq 0$, and if <i>uplo</i> = 'L', $n \leq m$.</p> <p>When $n = 0$, <code>?lantr</code> is set to zero.</p>
<i>a</i>	<p>Array, size at least $\max(1, lda \cdot n)$ for column major and $\max(1, lda \cdot m)$ for row major layout.</p> <p>The trapezoidal matrix A (A is triangular if $m = n$).</p> <p>If <i>uplo</i> = 'U', the leading m-by-n upper trapezoidal part of the array a contains the upper trapezoidal matrix, and the strictly lower triangular part of A is not referenced.</p>

If *uplo* = 'L', the leading *m*-by-*n* lower trapezoidal part of the array *a* contains the lower trapezoidal matrix, and the strictly upper triangular part of *A* is not referenced. Note that when *diag* = 'U', the diagonal elements of *A* are not referenced and are assumed to be one.

lda

The leading dimension of the array *a*.

lda ≥ max(*m*, 1) for column major layout and ≥max(1, *n*) for row major layout.

LAPACKE_set_nancheck

Turns NaN checking off or on

```
LAPACKE_set_nancheck(int flag);
```

Description

The routine sets a value for the LAPACKE NaN checking flag, which indicates whether or not LAPACKE routines check input matrices for NaNs.

Input Parameters

flag

If *flag*= 0, NaN checking is turned OFF. Otherwise, it is turned ON.

LAPACKE_get_nancheck

Gets the current NaN checking flag, which indicates whether NaN checking has been turned off or on.

```
int flag = LAPACKE_get_nancheck ();
```

Description

The function returns the current value for the LAPACKE NaN checking flag, which indicates whether or not LAPACKE routines check input matrices for NaNs.

Return Value

An integer value is returned which indicates the current NaN checking status.

The returned flag value is either 0 (OFF) or 1 (ON), even though any integer value can be used as an input parameter for LAPACKE_set_nancheck.

For example, the following code turns on NaN checking:

```
LAPACKE_set_nancheck(100);
int flag = LAPACKE_get_nancheck(); // flag==1, not 100.
```

?lapmr

Rearranges rows of a matrix as specified by a permutation vector.

Syntax

```
lapack_int LAPACKE_slapmr (int matrix_layout, lapack_logical forwr, lapack_int m,
lapack_int n, float* x, lapack_int ldx, lapack_int * k);
```

```
lapack_int LAPACKE_dlapmr (int matrix_layout, lapack_logical forwr, lapack_int m,
lapack_int n, double* x, lapack_int ldx, lapack_int * k);
```

```
lapack_int LAPACKE_clapmr (int matrix_layout, lapack_logical forwrd, lapack_int m,
lapack_int n, lapack_complex_float* x, lapack_int ldx, lapack_int * k);
```

```
lapack_int LAPACKE_zlapmr (int matrix_layout, lapack_logical forwrd, lapack_int m,
lapack_int n, lapack_complex_double* x, lapack_int ldx, lapack_int * k);
```

Include Files

- mkl.h

Description

The `?lapmr` routine rearranges the rows of the m -by- n matrix X as specified by the permutation $k[0]$, $k[1]$, ..., $k[m-1]$ of the integers $1, \dots, m$.

If *forwrd* is true, forward permutation:

$X(k[i-1], :)$ is moved to $X(i, :)$ for $i = 1, 2, \dots, m$.

If *forwrd* is false, backward permutation:

$X(i, :)$ is moved to $X(k[i-1], :)$ for $i = 1, 2, \dots, m$.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>forwrd</i>	If <i>forwrd</i> is true, forward permutation. If <i>forwrd</i> is false, backward permutation.
<i>m</i>	The number of rows of the matrix X . $m \geq 0$.
<i>n</i>	The number of columns of the matrix X . $n \geq 0$.
<i>x</i>	Array, size at least $\max(1, \text{ldx} * n)$ for column major and $\max(1, \text{ldx} * m)$ for row major layout. On entry, the m -by- n matrix X .
<i>ldx</i>	The leading dimension of the array X , $\text{ldx} \geq \max(1, m)$ for column major layout and $\text{ldx} \geq \max(1, n)$ for row major layout.
<i>k</i>	Array, size (m). On entry, k contains the permutation vector and is used as internal workspace.

Output Parameters

<i>x</i>	On exit, x contains the permuted matrix X .
<i>k</i>	On exit, k is reset to its original value.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

See Also

?lapmt

Performs a forward or backward permutation of the columns of a matrix.

Syntax

```
lapack_int LAPACKE_slapmt (int matrix_layout, lapack_logical forwr, lapack_int m,
lapack_int n, float * x, lapack_int ldx, lapack_int * k);

lapack_int LAPACKE_dlapmt (int matrix_layout, lapack_logical forwr, lapack_int m,
lapack_int n, double * x, lapack_int ldx, lapack_int * k);

lapack_int LAPACKE_clapmt (int matrix_layout, lapack_logical forwr, lapack_int m,
lapack_int n, lapack_complex_float * x, lapack_int ldx, lapack_int * k);

lapack_int LAPACKE_zlapmt (int matrix_layout, lapack_logical forwr, lapack_int m,
lapack_int n, lapack_complex_double * x, lapack_int ldx, lapack_int * k);
```

Include Files

- mkl.h

Description

The routine ?lapmt rearranges the columns of the m -by- n matrix X as specified by the permutation $k[i - 1]$ for $i = 1, \dots, n$.

If $forwr \neq 0$, forward permutation:

$X(*, k(j))$ is moved to $X(*, j)$ for $j = 1, 2, \dots, n$.

If $forwr = 0$, backward permutation:

$X(*, j)$ is moved to $X(*, k(j))$ for $j = 1, 2, \dots, n$.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>forwr</i>	If $forwr \neq 0$, forward permutation If $forwr = 0$, backward permutation
<i>m</i>	The number of rows of the matrix X . $m \geq 0$.
<i>n</i>	The number of columns of the matrix X . $n \geq 0$.
<i>x</i>	Array, size $ldx \times n$. On entry, the m -by- n matrix X .
<i>ldx</i>	The leading dimension of the array x , $ldx \geq \max(1, m)$.
<i>k</i>	Array, size (n) . On entry, k contains the permutation vector and is used as internal workspace.

Output Parameters

<i>x</i>	On exit, x contains the permuted matrix X .
<i>k</i>	On exit, k is reset to its original value.

See Also

[?lapmr](#)

?lapy2

Returns $\sqrt{x^2+y^2}$.

Syntax

```
float LAPACKE_slapy2 (floatx, floaty);
double LAPACKE_dlapy2 (doublex, doubley);
```

Include Files

- `mkl.h`

Description

The function `?lapy2` returns $\sqrt{x^2+y^2}$, avoiding unnecessary overflow or harmful underflow.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

x, y Specify the input values *x* and *y*.

Return Values

The function returns a value *val*.

If *val*=-1D0, the first argument was NaN.

If *val*=-2D0, the second argument was NaN.

?lapy3

Returns $\sqrt{x^2+y^2+z^2}$.

Syntax

```
float LAPACKE_slapy3 (floatx, floaty, floatz);
double LAPACKE_dlapy3 (double x, doubley, doublez);
```

Include Files

- `mkl.h`

Description

The function `?lapy3` returns $\sqrt{x^2+y^2+z^2}$, avoiding unnecessary overflow or harmful underflow.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

x, y, z Specify the input values *x*, *y* and *z*.

Return Values

This function returns a value *val*.

If *val* = -1D0, the first argument was NaN.

If *val* = -2D0, the second argument was NaN.

If *val* = -3D0, the third argument was NaN.

?laran

Returns a random real number from a uniform distribution.

Syntax

```
float slaran (lapack_int *iseed);
double dlaran (lapack_int *iseed);
```

Description

The ?laran routine returns a random real number from a uniform (0,1) distribution. This routine uses a multiplicative congruential method with modulus 2^{48} and multiplier 33952834046453. 48-bit integers are stored in four integer array elements with 12 bits per element. Hence the routine is portable across machines with integers of 32 bits or more.

Input Parameters

<i>iseed</i>	Array, size 4. On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and <i>iseed</i> [3] must be odd.
--------------	--

Output Parameters

<i>iseed</i>	On exit, the seed is updated.
--------------	-------------------------------

Return Values

The function returns a random number.

?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```
lapack_int LAPACKE_slarfb (int matrix_layout , char side , char trans , char direct ,
char storev , lapack_int m , lapack_int n , lapack_int k , const float * v , lapack_int
ldv , const float * t , lapack_int ldt , float * c , lapack_int ldc );

lapack_int LAPACKE_dlarfb (int matrix_layout , char side , char trans , char direct ,
char storev , lapack_int m , lapack_int n , lapack_int k , const double * v ,
lapack_int ldv , const double * t , lapack_int ldt , double * c , lapack_int
ldc ); lapack_int LAPACKE_clarfb (int matrix_layout , char side , char trans , char
direct , char storev , lapack_int m , lapack_int n , lapack_int k , const
lapack_complex_float * v , lapack_int ldv , const lapack_complex_float * t , lapack_int
ldt , lapack_complex_float * c , lapack_int ldc );
```

```
lapack_int LAPACKE_zlarfb (int matrix_layout , char side , char trans , char direct ,
char storev , lapack_int m , lapack_int n , lapack_int k , const lapack_complex_double
* v , lapack_int ldv , const lapack_complex_double * t , lapack_int ldt ,
lapack_complex_double * c , lapack_int ldc );
```

Include Files

- mkl.h

Description

The real flavors of the routine `?larfb` apply a real block reflector H or its transpose H^T to a real m -by- n matrix C from either left or right.

The complex flavors of the routine `?larfb` apply a complex block reflector H or its conjugate transpose H^H to a complex m -by- n matrix C from either left or right.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>side</i>	<p>If <i>side</i> = 'L': apply H or H^T for real flavors and H or H^H for complex flavors from the left.</p> <p>If <i>side</i> = 'R': apply H or H^T for real flavors and H or H^H for complex flavors from the right.</p>
<i>trans</i>	<p>If <i>trans</i> = 'N': apply H (No transpose).</p> <p>If <i>trans</i> = 'C': apply H^H (Conjugate transpose).</p> <p>If <i>trans</i> = 'T': apply H^T (Transpose).</p>
<i>direct</i>	<p>Indicates how H is formed from a product of elementary reflectors</p> <p>If <i>direct</i> = 'F': $H = H(1) * H(2) * \dots * H(k)$ (forward)</p> <p>If <i>direct</i> = 'B': $H = H(k) * \dots * H(2) * H(1)$ (backward)</p>
<i>storev</i>	<p>Indicates how the vectors which define the elementary reflectors are stored:</p> <p>If <i>storev</i> = 'C': Column-wise</p> <p>If <i>storev</i> = 'R': Row-wise</p>
<i>m</i>	The number of rows of the matrix C .
<i>n</i>	The number of columns of the matrix C .
<i>k</i>	The order of the matrix T (equal to the number of elementary reflectors whose product defines the block reflector).
<i>v</i>	The size limitations depend on values of parameters <i>storev</i> and <i>side</i> as described in the following table:

<i>storev</i> = C		<i>storev</i> = R	
<i>side</i> = L	<i>side</i> = R	<i>side</i> = L	<i>side</i> = R

Column major	$\max(1, ldv * k)$	$\max(1, ldv * k)$	$\max(1, ldv * m)$	$\max(1, ldv * n)$
Row major	$\max(1, ldv * m)$	$\max(1, ldv * n)$	$\max(1, ldv * k)$	$\max(1, ldv * k)$

The matrix v . See *Application Notes* below.

ldv

The leading dimension of the array v . It should satisfy the following conditions:

		$storev = C$		$storev = R$	
		$side = L$	$side = R$	$side = L$	$side = R$
Column major		$\max(1, m)$	$\max(1, n)$	$\max(1, k)$	$\max(1, k)$
Row major		$\max(1, k)$	$\max(1, k)$	$\max(1, m)$	$\max(1, n)$

t

Array, size at least $\max(1, ldt * k)$.

Contains the triangular k -by- k matrix T in the representation of the block reflector.

ldt

The leading dimension of the array t .

$ldt \geq k$.

c

Array, size at least $\max(1, ldc * n)$ for column major layout and $\max(1, ldc * m)$ for row major layout.

On entry, the m -by- n matrix C .

ldc

The leading dimension of the array c .

$ldc \geq \max(1, m)$ for column major layout and $ldc \geq \max(1, n)$ for row major layout.

Output Parameters

c

On exit, c is overwritten by the product of the following:

- H^*C , or $H^T C$, or C^*H , or $C^H H^T$ for real flavors
- H^*C , or $H^H C$, or C^*H , or $C^H H^H$ for complex flavors

Return Values

This function returns a value *info*.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = -1011$, memory allocation error occurred.

Application Notes

The shape of the matrix V and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and *storev* = 'C': *direct* = 'F' and *storev* = 'R':

$$\begin{bmatrix} 1 & & & & \\ v_1 & 1 & & & \\ v_1 & v_2 & 1 & & \\ v_1 & v_2 & v_3 & & \\ v_1 & v_2 & v_3 & & \end{bmatrix}$$

$$\begin{bmatrix} 1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & v_2 & v_2 & v_2 \\ & & 1 & v_3 & v_3 \end{bmatrix}$$

direct = 'B' and *storev* = 'C': *direct* = 'B' and *storev* = 'R':

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ & 1 & v_3 \\ & & 1 \end{bmatrix}$$

$$\begin{bmatrix} v_1 & v_1 & 1 & & \\ v_2 & v_2 & v_2 & 1 & \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
lapack_int LAPACKE_slarfg (lapack_int n , float * alpha , float * x , lapack_int incx , float * tau );
```

```
lapack_int LAPACKE_dlarfg (lapack_int n , double * alpha , double * x , lapack_int incx , double * tau );
```

```
lapack_int LAPACKE_clarfg (lapack_int n , lapack_complex_float * alpha , lapack_complex_float * x , lapack_int incx , lapack_complex_float * tau );
```

```
lapack_int LAPACKE_zlarfg (lapack_int n , lapack_complex_double * alpha , lapack_complex_double * x , lapack_int incx , lapack_complex_double * tau );
```

Include Files

- mkl.h

Description

The routine ?larfg generates a real/complex elementary reflector H of order n , such that

$$H^* \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, \quad H^T H = I,$$

for real flavors and

$$H^X * \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, H^X * H = I,$$

for complex flavors,

where α and β are scalars (with β real for all flavors), and x is an $(n-1)$ -element real/complex vector. H is represented in the form

$$H = I - \tau a u^* \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^T \end{bmatrix}$$

for real flavors and

$$H = I - \tau a u^* \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^H \end{bmatrix}$$

for complex flavors,

where $\tau a u$ is a real/complex scalar and v is a real/complex $(n-1)$ -element vector, respectively. Note that for `clarfg/zlarfg`, H is not Hermitian.

If the elements of x are all zero (and, for complex flavors, α is real), then $\tau a u = 0$ and H is taken to be the unit matrix.

Otherwise, $1 \leq \tau a u \leq 2$ (for real flavors), or

$1 \leq \text{Re}(\tau a u) \leq 2$ and $\text{abs}(\tau a u - 1) \leq 1$ (for complex flavors).

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

n	The order of the elementary reflector.
α	
x	Array, size $(1+(n-2)*\text{abs}(\text{incx}))$. On entry, the vector x .
incx	The increment between elements of x . $\text{incx} > 0$.

Output Parameters

α	On exit, it is overwritten with the value β .
x	On exit, it is overwritten with the vector v .
$\tau a u$	

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -2`, α is NaN

If `info = -3`, array x contains NaN components.

?larft

Forms the triangular factor T of a block reflector $H = I - V^*T^*V^{**}H$.

Syntax

```
lapack_int LAPACKE_slarft (int matrix_layout , char direct , char storev , lapack_int
n , lapack_int k , const float * v , lapack_int ldv , const float * tau , float * t ,
lapack_int ldt );

lapack_int LAPACKE_dlarft (int matrix_layout , char direct , char storev , lapack_int
n , lapack_int k , const double * v , lapack_int ldv , const double * tau , double * t ,
lapack_int ldt );

lapack_int LAPACKE_clarft (int matrix_layout , char direct , char storev , lapack_int
n , lapack_int k , const lapack_complex_float * v , lapack_int ldv , const
lapack_complex_float * tau , lapack_complex_float * t , lapack_int ldt );

lapack_int LAPACKE_zlarft (int matrix_layout , char direct , char storev , lapack_int
n , lapack_int k , const lapack_complex_double * v , lapack_int ldv , const
lapack_complex_double * tau , lapack_complex_double * t , lapack_int ldt );
```

Include Files

- mkl.h

Description

The routine ?larft forms the triangular factor T of a real/complex block reflector H of order n , which is defined as a product of k elementary reflectors.

If $direct = 'F'$, $H = H(1)*H(2)* \dots *H(k)$ and T is upper triangular;

If $direct = 'B'$, $H = H(k)* \dots *H(2)*H(1)$ and T is lower triangular.

If $storev = 'C'$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th column of the array v , and $H = I - V^*T^*V^T$ (for real flavors) or $H = I - V^*T^*V^H$ (for complex flavors) .

If $storev = 'R'$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th row of the array v , and $H = I - V^T*T^*V$ (for real flavors) or $H = I - V^H*T^*V$ (for complex flavors).

Input Parameters

A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>direct</i>	<p>Specifies the order in which the elementary reflectors are multiplied to form the block reflector:</p> <p>= 'F': $H = H(1)*H(2)* \dots *H(k)$ (forward)</p> <p>= 'B': $H = H(k)* \dots *H(2)*H(1)$ (backward)</p>
<i>storev</i>	<p>Specifies how the vectors which define the elementary reflectors are stored (see also <i>Application Notes</i> below):</p> <p>= 'C': column-wise</p> <p>= 'R': row-wise.</p>
<i>n</i>	The order of the block reflector H . $n \geq 0$.

k The order of the triangular factor T (equal to the number of elementary reflectors). $k \geq 1$.

v The size limitations depend on values of parameters *storev* and *side* as described in the following table:

	<i>storev</i> = C	<i>storev</i> = R
Column major	$\max(1, ldv * k)$	$\max(1, ldv * n)$
Row major	$\max(1, ldv * n)$	$\max(1, ldv * k)$

The matrix *v*. See *Application Notes* below.

ldv The leading dimension of the array *v*.

If *storev* = 'C', $ldv \geq \max(1, n)$ for column major and $ldv \geq \max(1, k)$ for row major;

if *storev* = 'R', $ldv \geq k$ for column major and $ldv \geq \max(1, n)$ for row major.

tau Array, size (*k*). *tau*[*i*-1] must contain the scalar factor of the elementary reflector $H(i)$.

ldt The leading dimension of the output array *t*. $ldt \geq k$.

Output Parameters

t Array, size *ldt* * *k*. The *k*-by-*k* triangular factor T of the block reflector. If *direct* = 'F', T is upper triangular; if *direct* = 'B', T is lower triangular. The rest of the array is not used.

v The matrix V .

Application Notes

The shape of the matrix V and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and *storev* = 'C': *direct* = 'F' and *storev* = 'R':

$$\begin{bmatrix} 1 & & & & \\ v_1 & 1 & & & \\ v_1 & v_2 & 1 & & \\ v_1 & v_2 & v_3 & & \\ v_1 & v_2 & v_3 & & \end{bmatrix}$$

$$\begin{bmatrix} 1 & v_1 & v_1 & v_1 & v_1 \\ & 1 & v_2 & v_2 & v_2 \\ & & 1 & v_3 & v_3 \end{bmatrix}$$

direct = 'B' and storev = 'C': *direct = 'B' and storev = 'R':*

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ & 1 & v_3 \\ & & 1 \end{bmatrix}$$

$$\begin{bmatrix} v_1 & v_1 & 1 & & \\ v_2 & v_2 & v_2 & 1 & \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfx

Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order less than or equal to 10.

Syntax

```
lapack_int LAPACKE_slarfx (int matrix_layout , char side , lapack_int m , lapack_int
n , const float * v , float tau , float * c , lapack_int ldc , float * work );
```

```
lapack_int LAPACKE_dlarfx (int matrix_layout , char side , lapack_int m , lapack_int
n , const double * v , double tau , double * c , lapack_int ldc , double * work );
```

```
lapack_int LAPACKE_clarfx (int matrix_layout , char side , lapack_int m , lapack_int
n , const lapack_complex_float * v , lapack_complex_float tau , lapack_complex_float *
c , lapack_int ldc , lapack_complex_float * work );
```

```
lapack_int LAPACKE_zlarfx (int matrix_layout , char side , lapack_int m , lapack_int
n , const lapack_complex_double * v , lapack_complex_double tau , lapack_complex_double
* c , lapack_int ldc , lapack_complex_double * work );
```

Include Files

- mkl.h

Description

The routine ?larfx applies a real/complex elementary reflector H to a real/complex m -by- n matrix C , from either the left or the right.

H is represented in the following forms:

- $H = I - \tau v v^T$, where τ is a real scalar and v is a real vector.
- $H = I - \tau v v^H$, where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then H is taken to be the unit matrix.

Input Parameters

A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

side If *side* = 'L': form H^*C
 If *side* = 'R': form C^*H .

<i>m</i>	The number of rows of the matrix <i>C</i> .
<i>n</i>	The number of columns of the matrix <i>C</i> .
<i>v</i>	Array, size (<i>m</i>) if <i>side</i> = 'L' or (<i>n</i>) if <i>side</i> = 'R'. The vector <i>v</i> in the representation of <i>H</i> .
<i>tau</i>	The value <i>tau</i> in the representation of <i>H</i> .
<i>c</i>	Array, size at least max(1, <i>ldc</i> * <i>n</i>) for column major layout and max (1, <i>ldc</i> * <i>m</i>) for row major layout. On entry, the <i>m</i> -by- <i>n</i> matrix <i>C</i> .
<i>ldc</i>	The leading dimension of the array <i>c</i> . <i>lda</i> ≥ (1, <i>m</i>).
<i>work</i>	Workspace array, size (<i>n</i>) if <i>side</i> = 'L' or (<i>m</i>) if <i>side</i> = 'R'. <i>work</i> is not referenced if <i>H</i> has order < 11.

Output Parameters

<i>c</i>	On exit, <i>C</i> is overwritten by the matrix H^*C if <i>side</i> = 'L', or C^*H if <i>side</i> = 'R'.
----------	---

?large

Pre- and post-multiplies a real general matrix with a random orthogonal matrix.

Syntax

```
void slarge (lapack_int *n, float *a, lapack_int *lda, lapack_int *iseed, float * work,
lapack_int *info);

void dlarge (lapack_int *n, double *a, lapack_int *lda, lapack_int *iseed, double *
work, lapack_int *info);

void clarge (lapack_int *n, lapack_complex *a, lapack_int *lda, lapack_int *iseed,
lapack_complex * work, lapack_int *info);

void zlarge (lapack_int *n, lapack_complex_double *a, lapack_int *lda, lapack_int
*iseed, lapack_complex_double * work, lapack_int *info);
```

Include Files

- mkl.h

Description

The routine ?large pre- and post-multiplies a general *n*-by-*n* matrix *A* with a random orthogonal or unitary matrix: $A = U^*D^*U^T$.

Input Parameters

<i>n</i>	The order of the matrix <i>A</i> . $n \geq 0$
<i>a</i>	Array, size <i>lda</i> by <i>n</i> . On entry, the original <i>n</i> -by- <i>n</i> matrix <i>A</i> .
<i>lda</i>	The leading dimension of the array <i>a</i> . $lda \geq n$.
<i>iseed</i>	Array, size 4. On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and <i>iseed</i> [3] must be odd.
<i>work</i>	Workspace array, size $2*n$.

Output Parameters

<i>a</i>	On exit, <i>A</i> is overwritten by $U*A*U'$ for some random orthogonal matrix <i>U</i> .
<i>iseed</i>	On exit, the seed is updated.
<i>info</i>	If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, the <i>i</i> -th parameter had an illegal value.

?larnd

Returns a random real number from a uniform or normal distribution.

Syntax

```
float slarnd (lapack_int *idist, lapack_int *iseed);
double dlarnd (lapack_int *idist, lapack_int *iseed);
```

The data types for complex variations depend on whether or not the application links with Gnu Fortran (gfortran) libraries.

For non-gfortran (libmkl_intel_*) interface libraries:

```
void clarnd (lapack_complex_float *res, lapack_int *idist, lapack_int *iseed);
void zlarnd (lapack_complex_double *res, lapack_int *idist, lapack_int *iseed);
```

For gfortran (libmkl_gf_*) interface libraries:

```
lapack_complex_float clarnd (lapack_int *idist, lapack_int *iseed);
lapack_complex_double zlarnd (lapack_int *idist, lapack_int *iseed);
```

To understand the difference between the non-gfortran and gfortran interfaces and when to use each of them, see Dynamic Libraries in the lib/intel64 Directory in the *oneAPI Math Kernel Library Developer Guide*.

Include Files

- mkl.h

Description

The routine ?larnd returns a random number from a uniform or normal distribution.

Input Parameters

<i>idist</i>	<p>Specifies the distribution of the random numbers. For <code>slanrd</code> and <code>dlanrd</code>:</p> <ul style="list-style-type: none"> = 1: uniform (0,1) = 2: uniform (-1,1) = 3: normal (0,1). <p>For <code>clarnd</code> and <code>zlanrd</code>:</p> <ul style="list-style-type: none"> = 1: real and imaginary parts each uniform (0,1) = 2: real and imaginary parts each uniform (-1,1) = 3: real and imaginary parts each normal (0,1) = 4: uniformly distributed on the disc $\text{abs}(z) \leq 1$ = 5: uniformly distributed on the circle $\text{abs}(z) = 1$
<i>iseed</i>	<p>Array, size 4.</p> <p>On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and <code>iseed[3]</code> must be odd.</p>

Output Parameters

<i>iseed</i>	On exit, the seed is updated.
--------------	-------------------------------

Return Values

The function returns a random number (for complex variations `libmkl_gf_*` interface layer/libraries return the result as the parameter `res`).

?larnv

Returns a vector of random numbers from a uniform or normal distribution.

Syntax

```
lapack_int LAPACKE_slarnv (lapack_int idist , lapack_int * iseed , lapack_int n , float
* x );

lapack_int LAPACKE_dlarnv (lapack_int idist , lapack_int * iseed , lapack_int n ,
double * x );

lapack_int LAPACKE_clarnv (lapack_int idist , lapack_int * iseed , lapack_int n ,
lapack_complex_float * x );

lapack_int LAPACKE_zlarnv (lapack_int idist , lapack_int * iseed , lapack_int n ,
lapack_complex_double * x );
```

Include Files

- `mkl.h`

Description

The routine `?larnv` returns a vector of n random real/complex numbers from a uniform or normal distribution.

This routine calls the auxiliary routine `?laruv` to generate random real numbers from a uniform (0,1) distribution, in batches of up to 128 using vectorisable code. The Box-Muller method is used to transform numbers from a uniform to a normal distribution.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<code>idist</code>	Specifies the distribution of the random numbers: for <code>slarnv</code> and <code>dlarnv</code> : = 1: uniform (0,1) = 2: uniform (-1,1) = 3: normal (0,1). for <code>clarv</code> and <code>zlarv</code> : = 1: real and imaginary parts each uniform (0,1) = 2: real and imaginary parts each uniform (-1,1) = 3: real and imaginary parts each normal (0,1) = 4: uniformly distributed on the disc $\text{abs}(z) < 1$ = 5: uniformly distributed on the circle $\text{abs}(z) = 1$
<code>iseed</code>	Array, size (4). On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and <code>iseed(4)</code> must be odd.
<code>n</code>	The number of random numbers to be generated.

Output Parameters

<code>x</code>	Array, size (n). The generated random numbers.
<code>iseed</code>	On exit, the seed is updated.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

?laror

Pre- or post-multiplies an m -by- n matrix by a random orthogonal/unitary matrix.

Syntax

```
void slaror (char *side, char *init, lapack_int *m, lapack_int *n, float *a, lapack_int
*lda, lapack_int *iseed, float *x, lapack_int *info);

void dlaror (char *side, char *init, lapack_int *m, lapack_int *n, double *a, lapack_int
*lda, lapack_int *iseed, double *x, lapack_int *info);

void claror (char *side, char *init, lapack_int *m, lapack_int *n, lapack_complex *a,
lapack_int *lda, lapack_int *iseed, lapack_complex *x, lapack_int *info);
```

```
void zlaror (char *side, char *init, lapack_int *m, lapack_int *n,
lapack_complex_double *a, lapack_int *lda, lapack_int *iseed, lapack_complex_double *x,
lapack_int *info);
```

Include Files

- mkl.h

Description

The routine `?laror` pre- or post-multiplies an m -by- n matrix A by a random orthogonal or unitary matrix U , overwriting A . A may optionally be initialized to the identity matrix before multiplying by U . U is generated using the method of G.W. Stewart (SIAM J. Numer. Anal. 17, 1980, 403-409).

Input Parameters

<i>side</i>	<p>Specifies whether A is multiplied by U on the left or right.</p> <p>for <code>slaror</code> and <code>dlaror</code>:</p> <p>If <i>side</i> = 'L', multiply A on the left (premultiply) by U.</p> <p>If <i>side</i> = 'R', multiply A on the right (postmultiply) by U^T.</p> <p>If <i>side</i> = 'C' or 'T', multiply A on the left by U and the right by U^T.</p> <p>for <code>claror</code> and <code>zlaror</code>:</p> <p>If <i>side</i> = 'L', multiply A on the left (premultiply) by U.</p> <p>If <i>side</i> = 'R', multiply A on the right (postmultiply) by $UC^>$.</p> <p>If <i>side</i> = 'C', multiply A on the left by U and the right by $UC^>$.</p> <p>If <i>side</i> = 'T', multiply A on the left by U and the right by U^T.</p>
<i>init</i>	<p>Specifies whether or not a should be initialized to the identity matrix.</p> <p>If <i>init</i> = 'I', initialize a to (a section of) the identity matrix before applying U.</p> <p>If <i>init</i> = 'N', no initialization. Apply U to the input matrix A.</p> <p><i>init</i> = 'I' generates square or rectangular orthogonal matrices:</p> <p>For $m = n$ and <i>side</i> = 'L' or 'R', the rows and the columns are orthogonal to each other.</p> <p>For rectangular matrices where $m < n$:</p> <ul style="list-style-type: none"> • If <i>side</i> = 'R', <code>?laror</code> produces a dense matrix in which rows are orthogonal and columns are not. • If <i>side</i> = 'L', <code>?laror</code> produces a matrix in which rows are orthogonal, first m columns are orthogonal, and remaining columns are zero. <p>For rectangular matrices where $m > n$:</p> <ul style="list-style-type: none"> • If <i>side</i> = 'L', <code>?laror</code> produces a dense matrix in which columns are orthogonal and rows are not. • If <i>side</i> = 'R', <code>?laror</code> produces a matrix in which columns are orthogonal, first m rows are orthogonal, and remaining rows are zero.
<i>m</i>	<p>The number of rows of A.</p>

n	The number of columns of A .
a	Array, size lda by n .
lda	The leading dimension of the array a . $lda \geq \max(1, m)$.
$iseed$	Array, size (4). On entry, specifies the seed of the random number generator. The array elements must be between 0 and 4095; if not they are reduced mod 4096. Also, $iseed[3]$ must be odd.
x	Workspace array, size $(3 * \max(m, n))$.

Value of $side$	Length of workspace
'L'	$2 * m + n$
'R'	$2 * n + m$
'C' or 'T'	$3 * n$

Output Parameters

a	On exit, overwritten by UA (if $side = 'L'$), by AU (if $side = 'R'$), by UAU^T (if $side = 'C'$ or $'T'$).
$iseed$	The values of $iseed$ are changed on exit, and can be used in the next call to continue the same random number sequence.
$info$	Array, size (4). For <code>slaror</code> and <code>dlaror</code> : If $info = 0$, the execution is successful. If $info < 0$, the i -th parameter had an illegal value. If $info = 1$, the random numbers generated by <code>?laror</code> are bad. For <code>claror</code> and <code>zlaror</code> : If $info = 0$, the execution is successful. If $info = -1$, $side$ is not 'L', 'R', 'C', or 'T'. If $info = -3$, if m is negative. If $info = -4$, if m is negative or if $side$ is 'C' or 'T' and n is not equal to m . If $info = -6$, if lda is less than m .

?larot

Applies a Givens rotation to two adjacent rows or columns.

Syntax

```
void slarot (lapack_logical *lrows, lapack_logical *ileft, lapack_logical *iright,
lapack_int *nl, float *c, float *s, float *a, lapack_int *lda, float *xleft, float
*xright);
```

```
void dlarot (lapack_logical *lrows, lapack_logical *ileft, lapack_logical *iright,
lapack_int *nl, double *c, double *s, double *a, lapack_int *lda, double *xleft, double
*xright);
```

```
void clarot (lapack_logical *lrows, lapack_logical *ileft, lapack_logical *iright,
lapack_int *nl, lapack_complex *c, lapack_complex *s, lapack_complex *a, lapack_int
*lda, lapack_complex *xleft, lapack_complex *xright);
```

```
void zlarot (lapack_logical *lrows, lapack_logical *ileft, lapack_logical *iright,
lapack_int *nl, lapack_complex_double *c, lapack_complex_double *s,
lapack_complex_double *a, lapack_int *lda, lapack_complex_double *xleft,
lapack_complex_double *xright);
```

Include Files

- mkl.h

Description

The routine `?larot` applies a Givens rotation to two adjacent rows or columns, where one element of the first or last column or row is stored in some format other than GE so that elements of the matrix may be used or modified for which no array element is provided.

One example is a symmetric matrix in SB format (bandwidth = 4), for which `uplo = 'L'`. Two adjacent rows will have the format:

```
row j :      C > C > C > C > C > . . . .
row j + 1 :  C > C > C > C > C > . . . .
```

'*' indicates elements for which storage is provided.

'.' indicates elements for which no storage is provided, but are not necessarily zero; their values are determined by symmetry.

' ' indicates elements which are required to be zero, and have no storage provided.

Those columns which have two '*' entries can be handled by `srot` (for `slarot` and `clarot`), or by `drot` (for `dlarot` and `zlarot`).

Those columns which have no '*' entries can be ignored, since as long as the Givens rotations are carefully applied to preserve symmetry, their values are determined.

Those columns which have one '*' have to be handled separately, by using separate variables *p* and *q* :

```
row j :      C > C > C > C > C > p. . . .
row j + 1 :  q C > C > C > C > C > . . . .
```

If element *p* is set correctly, `?larot` rotates the column and sets *p* to its new value. The next call to `?larot` rotates columns *j* and *j* + 1, and restore symmetry. The element *q* is zero at the beginning, and non-zero after the rotation. Later, rotations would presumably be chosen to zero *q* out.

Typical Calling Sequences: rotating the *i* -th and (*i* + 1) -st rows.

Input Parameters

<code>lrows</code>	<p>If <code>lrows = 1</code>, <code>?larot</code> rotates two rows.</p> <p>If <code>lrows = 0</code>, <code>?larot</code> rotates two columns.</p>
<code>lleft</code>	<p>If <code>lleft = 1</code>, <code>xleft</code> is used instead of the corresponding element of <code>a</code> for the first element in the second row (if <code>lrows = 0</code>) or column (if <code>lrows=1</code>).</p> <p>If <code>lleft = 0</code>, the corresponding element of <code>a</code> is used.</p>
<code>lright</code>	<p>If <code>lleft = 1</code>, <code>xright</code> is used instead of the corresponding element of <code>a</code> for the first element in the second row (if <code>lrows = 0</code>) or column (if <code>lrows=1</code>).</p> <p>If <code>lright = 0</code>, the corresponding element of <code>a</code> is used.</p>
<code>nl</code>	<p>The length of the rows (if <code>lrows=1</code>) or columns (if <code>lrows=1</code>) to be rotated.</p> <p>If <code>xleft</code> or <code>xright</code> are used, the columns or rows they are in should be included in <code>nl</code>, e.g., if <code>lleft = lright = 1</code>, then <code>nl</code> must be at least 2.</p> <p>The number of rows or columns to be rotated exclusive of those involving <code>xleft</code> and/or <code>xright</code> may not be negative, i.e., <code>nl</code> minus how many of <code>lleft</code> and <code>lright</code> are 1 must be at least zero; if not, <code>xerbla</code> is called.</p>
<code>c, s</code>	<p>Specify the Givens rotation to be applied.</p> <p>If <code>lrows = 1</code>, then the matrix</p> $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ <p>is applied from the left.</p> <p>If <code>lrows = 0</code>, then the transpose thereof is applied from the right.</p>
<code>a</code>	The array containing the rows or columns to be rotated. The first element of <code>a</code> should be the upper left element to be rotated.
<code>lda</code>	<p>The "effective" leading dimension of <code>a</code>.</p> <p>If <code>a</code> contains a matrix stored in GE or SY format, then this is just the leading dimension of <code>A</code>.</p> <p>If <code>a</code> contains a matrix stored in band (GB or SB) format, then this should be one less than the leading dimension used in the calling routine. Thus, if <code>a</code> in <code>?larot</code> is of size <code>lda*n</code>, then <code>a[(j - 1)*lda]</code> would be the <code>j</code>-th element in the first of the two rows to be rotated, and <code>a[(j - 1)*lda + 1]</code> would be the <code>j</code>-th in the second, regardless of how the array may be stored in the calling routine. <code>a</code> cannot be dimensioned, because for band format the row number may exceed <code>lda</code>, which is not legal FORTRAN.</p> <p>If <code>lrows = 1</code>, then <code>lda</code> must be at least 1, otherwise it must be at least <code>nl</code> minus the number of 1 values in <code>xleft</code> and <code>xright</code>.</p>
<code>xleft</code>	<p>If <code>lrows = 1</code>, <code>xleft</code> is used and modified instead of <code>a[1]</code> (if <code>lrows = 1</code>) or <code>a[lda + 1]</code> (if <code>lrows = 0</code>).</p>

xright If *lright* = 1, *xright* is used and modified instead of $a[(nl - 1)*lda]$ (if *lrows* = 1) or $a[nl - 1]$ (if *lrows* = 0).

Output Parameters

a On exit, modified array A.

?lartgp

Generates a plane rotation.

Syntax

```
lapack_int LAPACKE_slartgp (float f, float g, float* cs, float* sn, float* r);
lapack_int LAPACKE_dlartgp (double f, double g, double* cs, double* sn, double* r);
```

Include Files

- mkl.h

Description

The routine generates a plane rotation so that

$$\begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $cs^2 + sn^2 = 1$

This is a slower, more accurate version of the BLAS Level 1 routine `?rotg`, except for the following differences:

- *f* and *g* are unchanged on return.
- If *g*=0, then *cs*=(+/-)1 and *sn*=0.
- If *f*=0 and *g*≠0, then *cs*=0 and *sn*=(+/-)1.

The sign is chosen so that $r \geq 0$.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

f, g The first and second component of the vector to be rotated.

Output Parameters

cs The cosine of the rotation.

sn The sine of the rotation.

r The nonzero component of the rotated vector.

Return Values

If $info = 0$, the execution is successful.

If $info = -1$, f is NaN.

If $info = -2$, g is NaN.

See Also

[?rotg](#)

[?lartgs](#)

?lartgs

Generates a plane rotation designed to introduce a bulge in implicit QR iteration for the bidiagonal SVD problem.

Syntax

```
lapack_int LAPACKE_slartgs (floatx, floaty, floatsigma, float* cs, float* sn);
```

```
lapack_int LAPACKE_dlartgs (doublex, doubley, doublesigma, double* cs, double* sn);
```

Include Files

- `mkl.h`

Description

The routine generates a plane rotation designed to introduce a bulge in Golub-Reinsch-style implicit QR iteration for the bidiagonal SVD problem. x and y are the top-row entries, and σ is the shift. The computed cs and sn define a plane rotation that satisfies the following:

$$\begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix} \cdot \begin{bmatrix} x^2 - \sigma \\ x * y \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

with r nonnegative.

If $x^2 - \sigma$ and $x * y$ are 0, the rotation is by $\pi/2$

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

x, y The (1,1) and (1,2) entries of an upper bidiagonal matrix, respectively.

σ Shift

Output Parameters

cs The cosine of the rotation.

sn

The sine of the rotation.

Return Values

If *info* = 0, the execution is successful.

If *info* = - 1, *x* is NaN.

If *info* = - 2, *y* is NaN.

If *info* = - 3, *sigma* is NaN.

See Also

[?lartgp](#)

?lascl

Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from} .

Syntax

```
lapack_int LAPACKE_slascl (int matrix_layout, char type, lapack_int kl, lapack_int ku,
float cfrom, float cto, lapack_int m, lapack_int n, float * a, lapack_int lda);
```

```
lapack_int LAPACKE_dlascl (int matrix_layout, char type, lapack_int kl, lapack_int ku,
double cfrom, double cto, lapack_int m, lapack_int n, double * a, lapack_int lda);
```

```
lapack_int LAPACKE_clascl (int matrix_layout, char type, lapack_int kl, lapack_int ku,
float cfrom, float cto, lapack_int m, lapack_int n, lapack_complex_float * a,
lapack_int lda);
```

```
lapack_int LAPACKE_zlascl (int matrix_layout, char type, lapack_int kl, lapack_int ku,
double cfrom, double cto, lapack_int m, lapack_int n, lapack_complex_double * a,
lapack_int lda);
```

Include Files

- mkl.h

Description

The routine `?lascl` multiplies the *m*-by-*n* real/complex matrix *A* by the real scalar c_{to}/c_{from} . The operation is performed without over/underflow as long as the final result $c_{to} * A(i, j) / c_{from}$ does not over/underflow.

type specifies that *A* may be full, upper triangular, lower triangular, upper Hessenberg, or banded.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

type

This parameter specifies the storage type of the input matrix.

= 'G': *A* is a full matrix.

= 'L': *A* is a lower triangular matrix.

= 'U': *A* is an upper triangular matrix.

= 'H': *A* is an upper Hessenberg matrix.

= 'B': A is a symmetric band matrix with lower bandwidth kl and upper bandwidth ku and with the only the lower half stored

= 'Q': A is a symmetric band matrix with lower bandwidth kl and upper bandwidth ku and with the only the upper half stored.

= 'Z': A is a band matrix with lower bandwidth kl and upper bandwidth ku . See description of the `?gbtrf` function for storage details.

<code>kl</code>	The lower bandwidth of A . Referenced only if <code>type = 'B', 'Q' or 'Z'</code> .
<code>ku</code>	The upper bandwidth of A . Referenced only if <code>type = 'B', 'Q' or 'Z'</code> .
<code>cfrom, cto</code>	The matrix A is multiplied by <code>cto/cfrom</code> . $A(i, j)$ is computed without over/underflow if the final result $cto * A(i, j) / cfrom$ can be represented without over/underflow. <code>cfrom</code> must be nonzero.
<code>m</code>	The number of rows of the matrix A . $m \geq 0$.
<code>n</code>	The number of columns of the matrix A . $n \geq 0$.
<code>a</code>	Array, size $(lda * n)$. The matrix to be multiplied by <code>cto/cfrom</code> . See <code>type</code> for the storage type.
<code>lda</code>	The leading dimension of the array <code>a</code> . $lda \geq \max(1, m)$.

Output Parameters

<code>a</code>	The multiplied matrix A .
<code>info</code>	If <code>info = 0</code> - successful exit If <code>info = -i < 0</code> , the i -th argument had an illegal value.

See Also

[?gbtrf](#)

?lasd0

Computes the singular values of a real upper bidiagonal n -by- m matrix B with diagonal d and off-diagonal e . Used by `?bdsdc`.

Syntax

```
void slasd0( lapack_int *n, lapack_int *sqre, float *d, float *e, float *u, lapack_int
*ldu, float *vt, lapack_int *ldvt, lapack_int *smlsiz, lapack_int *iwork, float *work,
lapack_int *info );

void dlasd0( lapack_int *n, lapack_int *sqre, double *d, double *e, double *u,
lapack_int *ldu, double *vt, lapack_int *ldvt, lapack_int *smlsiz, lapack_int *iwork,
double *work, lapack_int *info );
```

Include Files

- `mkl.h`

Description

Using a divide and conquer approach, the routine `?lasd0` computes the singular value decomposition (SVD) of a real upper bidiagonal n -by- m matrix B with diagonal d and offdiagonal e , where $m = n + sqre$.

The algorithm computes orthogonal matrices U and VT such that $B = U^* S^* VT$. The singular values S are overwritten on d .

The related subroutine `?lasda` computes only the singular values, and optionally, the singular vectors in compact form.

Input Parameters

<code>n</code>	On entry, the row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array d .
<code>sqre</code>	Specifies the column dimension of the bidiagonal matrix. If <code>sqre = 0</code> : the bidiagonal matrix has column dimension $m = n$. If <code>sqre = 1</code> : the bidiagonal matrix has column dimension $m = n+1$.
<code>d</code>	Array, <code>DIMENSION (n)</code> . On entry, d contains the main diagonal of the bidiagonal matrix.
<code>e</code>	Array, <code>DIMENSION (m-1)</code> . Contains the subdiagonal entries of the bidiagonal matrix. On exit, e is destroyed.
<code>ldu</code>	On entry, leading dimension of the output array u .
<code>ldvt</code>	On entry, leading dimension of the output array vt .
<code>smlsiz</code>	On entry, maximum size of the subproblems at the bottom of the computation tree.
<code>iwork</code>	Workspace array, dimension must be at least $(8n)$.
<code>work</code>	Workspace array, dimension must be at least $(3m^2+2m)$.

Output Parameters

<code>d</code>	On exit d , If <code>info = 0</code> , contains singular values of the bidiagonal matrix.
<code>u</code>	Array, <code>DIMENSION</code> at least (ldu, n) . On exit, u contains the left singular vectors.
<code>vt</code>	Array, <code>DIMENSION</code> at least $(ldvt, m)$. On exit, vt^T contains the right singular vectors.
<code>info</code>	If <code>info = 0</code> : successful exit. If <code>info = -i < 0</code> , the i -th argument had an illegal value. If <code>info = 1</code> , a singular value did not converge.

`?lasd1`

Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by `?bdsdc`.

Syntax

```
void slasd1( lapack_int *nl, lapack_int *nr, lapack_int *sqre, float *d, float *alpha,
float *beta, float *u, lapack_int *ldu, float *vt, lapack_int *ldvt, lapack_int *idxq,
lapack_int *iwork, float *work, lapack_int *info );
```

```
void dlasd1( lapack_int *nl, lapack_int *nr, lapack_int *sqre, double *d, double *alpha,
double *beta, double *u, lapack_int *ldu, double *vt, lapack_int *ldvt, lapack_int
*idxq, lapack_int *iwork, double *work, lapack_int *info );
```

Include Files

- mkl.h

Description

The routine computes the SVD of an upper bidiagonal n -by- m matrix B , where $n = nl + nr + 1$ and $m = n + sqre$.

The routine `?lasd1` is called from `?lasd0`.

A related subroutine `?lasd7` handles the case in which the singular values (and the singular vectors in factored form) are desired.

`?lasd1` computes the SVD as follows:

$$VT = U(in) * \begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1^T & a & Z2^T & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix} * VT(in)$$

$= U(out) * (D(out) \ 0) * VT(out)$

where $Z^T = (Z1^T a Z2^T b) = u^T * VT^T$, and u is a vector of dimension m with $alpha$ and $beta$ in the $nl+1$ and $nl+2$ -th entries and zeros elsewhere; and the entry b is empty if $sqre = 0$.

The left singular vectors of the original matrix are stored in u , and the transpose of the right singular vectors are stored in vt , and the singular values are in d . The algorithm consists of three stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or when there are zeros in the Z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `?lasd2`.
2. The second stage consists of calculating the updated singular values. This is done by finding the square roots of the roots of the secular equation via the routine `?lasd4` (as called by `?lasd3`). This routine also calculates the singular vectors of the current problem.
3. The final stage consists of computing the updated singular vectors directly using the updated singular values. The singular vectors for the current problem are multiplied with the singular vectors from the overall problem.

Input Parameters

nl The row dimension of the upper block.

$nl \geq 1$.

<i>nr</i>	The row dimension of the lower block. $nr \geq 1$.
<i>sqre</i>	If <i>sqre</i> = 0: the lower block is an <i>nr</i> -by- <i>nr</i> square matrix. If <i>sqre</i> = 1: the lower block is an <i>nr</i> -by-(<i>nr</i> +1) rectangular matrix. The bidiagonal matrix has row dimension $n = nl + nr + 1$, and column dimension $m = n + sqre$.
<i>d</i>	Array, DIMENSION ($nl+nr+1$). $n = nl+nr+1$. On entry $d(1:nl, 1:nl)$ contains the singular values of the upper block; and $d(nl+2:n)$ contains the singular values of the lower block.
<i>alpha</i>	Contains the diagonal element associated with the added row.
<i>beta</i>	Contains the off-diagonal element associated with the added row.
<i>u</i>	Array, DIMENSION (<i>ldu</i> , <i>n</i>). On entry $u(1:nl, 1:nl)$ contains the left singular vectors of the upper block; $u(nl+2:n, nl+2:n)$ contains the left singular vectors of the lower block.
<i>ldu</i>	The leading dimension of the array <i>U</i> . $ldu \geq \max(1, n)$.
<i>vt</i>	Array, DIMENSION (<i>ldvt</i> , <i>m</i>), where $m = n + sqre$. On entry $vt(1:nl+1, 1:nl+1)^T$ contains the right singular vectors of the upper block; $vt(nl+2:m, nl+2:m)^T$ contains the right singular vectors of the lower block.
<i>ldvt</i>	The leading dimension of the array <i>vt</i> . $ldvt \geq \max(1, M)$.
<i>iwork</i>	Workspace array, DIMENSION ($4n$).
<i>work</i>	Workspace array, DIMENSION ($3m_2 + 2m$).

Output Parameters

<i>d</i>	On exit $d(1:n)$ contains the singular values of the modified matrix.
<i>alpha</i>	On exit, the diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(\text{D(I)})), I = 1, n$.
<i>beta</i>	On exit, the off-diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(\text{D(I)})), I = 1, n$.
<i>u</i>	On exit <i>u</i> contains the left singular vectors of the bidiagonal matrix.
<i>vt</i>	On exit vt^T contains the right singular vectors of the bidiagonal matrix.
<i>idxq</i>	Array, DIMENSION (<i>n</i>). Contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, $d(\text{idxq}(i = 1, n))$ will be in ascending order.
<i>info</i>	If <i>info</i> = 0: successful exit. If <i>info</i> = - <i>i</i> < 0, the <i>i</i> -th argument had an illegal value.

If *info* = 1, a singular value did not converge.

?lasd2

Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.

Syntax

```
void slasd2( lapack_int *nl, lapack_int *nr, lapack_int *sqre, lapack_int *k, float *d,
float *z, float *alpha, float *beta, float *u, lapack_int *ldu, float *vt, lapack_int
*ldvt, float *dsigma, float *u2, lapack_int *ldu2, float *vt2, lapack_int *ldvt2,
lapack_int *idxp, lapack_int *idx, lapack_int *idxq, lapack_int *coltyp, lapack_int
*info );
```

```
void dlasd2( lapack_int *nl, lapack_int *nr, lapack_int *sqre, lapack_int *k, double *d,
double *z, double *alpha, double *beta, double *u, lapack_int *ldu, double *vt,
lapack_int *ldvt, double *dsigma, double *u2, lapack_int *ldu2, double *vt2, lapack_int
*ldvt2, lapack_int *idxp, lapack_int *idx, lapack_int *idxq, lapack_int *coltyp,
lapack_int *info );
```

Include Files

- mkl.h

Description

The routine ?lasd2 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the Z vector. For each such occurrence the order of the related secular equation problem is reduced by one.

The routine ?lasd2 is called from ?lasd1.

Input Parameters

<i>nl</i>	The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	The row dimension of the lower block. $nr \geq 1$.
<i>sqre</i>	If <i>sqre</i> = 0): the lower block is an <i>nr</i> -by- <i>nr</i> square matrix If <i>sqre</i> = 1): the lower block is an <i>nr</i> -by-(<i>nr</i> +1) rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.
<i>d</i>	Array, DIMENSION (<i>n</i>). On entry <i>d</i> contains the singular values of the two submatrices to be combined.
<i>alpha</i>	Contains the diagonal element associated with the added row.
<i>beta</i>	Contains the off-diagonal element associated with the added row.

<i>u</i>	Array, <code>DIMENSION (ldu, n)</code> . On entry <i>u</i> contains the left singular vectors of two submatrices in the two square blocks with corners at (1,1), (<i>nl</i> , <i>nl</i>), and (<i>nl</i> +2, <i>nl</i> +2), (<i>n</i> , <i>n</i>).
<i>ldu</i>	The leading dimension of the array <i>u</i> . <i>ldu</i> ≥ <i>n</i> .
<i>ldu2</i>	The leading dimension of the output array <i>u2</i> . <i>ldu2</i> ≥ <i>n</i> .
<i>vt</i>	Array, <code>DIMENSION (ldvt, m)</code> . On entry, <i>vt</i> ^T contains the right singular vectors of two submatrices in the two square blocks with corners at (1,1), (<i>nl</i> +1, <i>nl</i> +1), and (<i>nl</i> +2, <i>nl</i> +2), (<i>m</i> , <i>m</i>).
<i>ldvt</i>	The leading dimension of the array <i>vt</i> . <i>ldvt</i> ≥ <i>m</i> .
<i>ldvt2</i>	The leading dimension of the output array <i>vt2</i> . <i>ldvt2</i> ≥ <i>m</i> .
<i>idxp</i>	Workspace array, <code>DIMENSION (n)</code> . This will contain the permutation used to place deflated values of <i>D</i> at the end of the array. On output <i>idxp</i> (2: <i>k</i>) points to the nondeflated <i>d</i> -values and <i>idxp</i> (<i>k</i> +1: <i>n</i>) points to the deflated singular values.
<i>idx</i>	Workspace array, <code>DIMENSION (n)</code> . This will contain the permutation used to sort the contents of <i>d</i> into ascending order.
<i>coltyp</i>	Workspace array, <code>DIMENSION (n)</code> . As workspace, this array contains a label that indicates which of the following types a column in the <i>u2</i> matrix or a row in the <i>vt2</i> matrix is: 1 : non-zero in the upper half only 2 : non-zero in the lower half only 3 : dense 4 : deflated.
<i>idxq</i>	Array, <code>DIMENSION (n)</code> . This parameter contains the permutation that separately sorts the two sub-problems in <i>D</i> in the ascending order. Note that entries in the first half of this permutation must first be moved one position backwards and entries in the second half must have <i>nl</i> +1 added to their values.

Output Parameters

<i>k</i>	Contains the dimension of the non-deflated matrix, This is the order of the related secular equation. $1 \leq k \leq n$.
<i>d</i>	On exit <i>D</i> contains the trailing (<i>n</i> - <i>k</i>) updated singular values (those which were deflated) sorted into increasing order.
<i>u</i>	On exit <i>u</i> contains the trailing (<i>n</i> - <i>k</i>) updated left singular vectors (those which were deflated) in its last <i>n</i> - <i>k</i> columns.
<i>z</i>	Array, <code>DIMENSION (n)</code> . On exit, <i>z</i> contains the updating row vector in the secular equation.
<i>dsigma</i>	Array, <code>DIMENSION (n)</code> . Contains a copy of the diagonal elements (<i>k</i> -1 singular values and one zero) in the secular equation.

<code>u2</code>	Array, <code>DIMENSION (ldu2, n)</code> . Contains a copy of the first $k-1$ left singular vectors which will be used by <code>?lasd3</code> in a matrix multiply (<code>?gemm</code>) to solve for the new left singular vectors. <code>u2</code> is arranged into four blocks. The first block contains a column with 1 at $nl+1$ and zero everywhere else; the second block contains non-zero entries only at and above nl ; the third contains non-zero entries only below $nl+1$; and the fourth is dense.
<code>vt</code>	On exit, vt^T contains the trailing $(n-k)$ updated right singular vectors (those which were deflated) in its last $n-k$ columns. In case <code>sqre = 1</code> , the last row of <code>vt</code> spans the right null space.
<code>vt2</code>	Array, <code>DIMENSION (ldvt2, n)</code> . $vt2^T$ contains a copy of the first k right singular vectors which will be used by <code>?lasd3</code> in a matrix multiply (<code>?gemm</code>) to solve for the new right singular vectors. <code>vt2</code> is arranged into three blocks. The first block contains a row that corresponds to the special 0 diagonal element in <i>sigma</i> ; the second block contains non-zeros only at and before $nl + 1$; the third block contains non-zeros only at and after $nl + 2$.
<code>idxc</code>	Array, <code>DIMENSION (n)</code> . This will contain the permutation used to arrange the columns of the deflated <i>u</i> matrix into three groups: the first group contains non-zero entries only at and above nl , the second contains non-zero entries only below $nl+2$, and the third is dense.
<code>coltyp</code>	On exit, it is an array of dimension 4, with <code>coltyp(i)</code> being the dimension of the <i>i</i> -th type columns.
<code>info</code>	If <code>info = 0</code>): successful exit If <code>info = -i < 0</code> , the <i>i</i> -th argument had an illegal value.

?lasd3

*Finds all square roots of the roots of the secular equation, as defined by the values in *D* and *Z*, and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.*

Syntax

```
void slasd3( lapack_int *nl, lapack_int *nr, lapack_int *sqre, lapack_int *k, float *d,
float *q, lapack_int *ldq, float *dsigma, float *u, lapack_int *ldu, float *u2,
lapack_int *ldu2, float *vt, lapack_int *ldvt, float *vt2, lapack_int *ldvt2,
lapack_int *idxc, lapack_int *ctot, float *z, lapack_int *info );

void dlasd3( lapack_int *nl, lapack_int *nr, lapack_int *sqre, lapack_int *k, double *d,
double *q, lapack_int *ldq, double *dsigma, double *u, lapack_int *ldu, double *u2,
lapack_int *ldu2, double *vt, lapack_int *ldvt, double *vt2, lapack_int *ldvt2,
lapack_int *idxc, lapack_int *ctot, double *z, lapack_int *info );
```

Include Files

- `mk1.h`

Description

The routine `?lasd3` finds all the square roots of the roots of the secular equation, as defined by the values in *D* and *Z*.

It makes the appropriate calls to `?lasd4` and then updates the singular vectors by matrix multiplication. The routine `?lasd3` is called from `?lasd1`.

Input Parameters

<code>nl</code>	<p>The row dimension of the upper block.</p> <p>$nl \geq 1$.</p>
<code>nr</code>	<p>The row dimension of the lower block.</p> <p>$nr \geq 1$.</p>
<code>sgre</code>	<p>If <code>sgre = 0</code>): the lower block is an nr-by-nr square matrix.</p> <p>If <code>sgre = 1</code>): the lower block is an nr-by-$(nr+1)$ rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sgre \geq n$ columns.</p>
<code>k</code>	The size of the secular equation, $1 \leq k \leq n$.
<code>q</code>	Workspace array, <code>DIMENSION</code> at least (ldq, k) .
<code>ldq</code>	<p>The leading dimension of the array <code>Q</code>.</p> <p>$ldq \geq k$.</p>
<code>dsigma</code>	Array, <code>DIMENSION</code> (k) . The first k elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.
<code>ldu</code>	<p>The leading dimension of the array <code>u</code>.</p> <p>$ldu \geq n$.</p>
<code>u2</code>	<p>Array, <code>DIMENSION</code> $(ldu2, n)$.</p> <p>The first k columns of this matrix contain the non-deflated left singular vectors for the split problem.</p>
<code>ldu2</code>	<p>The leading dimension of the array <code>u2</code>.</p> <p>$ldu2 \geq n$.</p>
<code>ldvt</code>	<p>The leading dimension of the array <code>vt</code>.</p> <p>$ldvt \geq n$.</p>
<code>vt2</code>	<p>Array, <code>DIMENSION</code> $(ldvt2, n)$.</p> <p>The first k columns of <code>vt2'</code> contain the non-deflated right singular vectors for the split problem.</p>
<code>ldvt2</code>	<p>The leading dimension of the array <code>vt2</code>.</p> <p>$ldvt2 \geq n$.</p>
<code>idxc</code>	<p>Array, <code>DIMENSION</code> (n).</p> <p>The permutation used to arrange the columns of <code>u</code> (and rows of <code>vt</code>) into three groups: the first group contains non-zero entries only at and above (or before) $nl + 1$; the second contains non-zero entries only at and below (or after) $nl+2$; and the third is dense. The first column of <code>u</code> and the row of</p>

vt are treated separately, however. The rows of the singular vectors found by ?lasd4 must be likewise permuted before the matrix multiplies can take place.

$ctot$ Array, DIMENSION (4). A count of the total number of the various types of columns in u (or rows in vt), as described in $idxc$.

The fourth column type is any column which has been deflated.

z Array, DIMENSION (k). The first k elements of this array contain the components of the deflation-adjusted updating row vector.

Output Parameters

d Array, DIMENSION (k). On exit the square roots of the roots of the secular equation, in ascending order.

u Array, DIMENSION (ldu, n).

The last $n - k$ columns of this matrix contain the deflated left singular vectors.

vt Array, DIMENSION ($ldvt, m$).

The last $m - k$ columns of vt' contain the deflated right singular vectors.

$vt2$ Destroyed on exit.

z Destroyed on exit.

$info$ If $info = 0$): successful exit.

If $info = -i < 0$, the i -th argument had an illegal value.

If $info = 1$, an singular value did not converge.

Application Notes

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

?lasd4

Computes the square root of the i -th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc.

Syntax

```
void slasd4( lapack_int *n, lapack_int *i, float *d, float *z, float *delta, float *rho,
float *sigma, float *work, lapack_int *info);
```

```
void dlasd4( lapack_int *n, lapack_int *i, double *d, double *z, double *delta, double
*rho, double *sigma, double *work, lapack_int *info);
```

Include Files

- mkl.h

Description

The routine computes the square root of the i -th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix whose entries are given as the squares of the corresponding entries in the array d , and that $0 \leq d(i) < d(j)$ for $i < j$ and that $\rho > 0$. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

$$\text{diag}(d) * \text{diag}(d) + \rho * Z * Z^T,$$

where the Euclidean norm of Z is equal to 1. The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

n	The length of all arrays.
i	The index of the eigenvalue to be computed. $1 \leq i \leq n$.
d	Array, DIMENSION (n). The original eigenvalues. They must be in order, $0 \leq d(i) < d(j)$ for $i < j$.
z	Array, DIMENSION (n). The components of the updating vector.
ρ	The scalar in the symmetric updating formula.
$work$	Workspace array, DIMENSION (n). If $n \neq 1$, $work$ contains $(d(j) + \sigma_i)$ in its j -th component. If $n = 1$, then $work(1) = 1$.

Output Parameters

δ	Array, DIMENSION (n). If $n \neq 1$, δ contains $(d(j) - \sigma_i)$ in its j -th component. If $n = 1$, then $\delta(1) = 1$. The vector δ contains the information necessary to construct the (singular) eigenvectors.
σ	The computed σ_i , the i -th updated eigenvalue.
$info$	= 0: successful exit > 0: If $info = 1$, the updating process failed.

?lasd5

Computes the square root of the i -th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix. Used by ?bdsdc.

Syntax

```
void slasd5( lapack_int *i, float *d, float *z, float *delta, float *rho, float *dsigma,
float *work );
```

```
void dlasd5( lapack_int *i, double *d, double *z, double *delta, double *rho, double
*dsigma, double *work );
```

Include Files

- mkl.h

Description

The routine computes the square root of the i -th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix $\text{diag}(d) * \text{diag}(d) + \rho * Z * Z^T$

The diagonal entries in the array d must satisfy $0 \leq d(i) < d(j)$ for $i < j$, ρ must be greater than 0, and that the Euclidean norm of the vector Z is equal to 1.

Input Parameters

i	The index of the eigenvalue to be computed. $i = 1$ or $i = 2$.
d	Array, dimension (2). The original eigenvalues, $0 \leq d(1) < d(2)$.
z	Array, dimension (2). The components of the updating vector.
ρ	The scalar in the symmetric updating formula.
$work$	Workspace array, dimension (2). Contains $(d(j) + \sigma_i)$ in its j -th component.

Output Parameters

δ	Array, dimension (2). Contains $(d(j) - \sigma_i)$ in its j -th component. The vector δ contains the information necessary to construct the eigenvectors.
σ_i	The computed σ_i , the i -th updated eigenvalue.

?lasd6

Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.

Syntax

```
void slasd6( lapack_int *icompq, lapack_int *nl, lapack_int *nr, lapack_int *sqre,
float *d, float *vf, float *vl, float *alpha, float *beta, lapack_int *idxq, lapack_int
*perm, lapack_int *givptr, lapack_int *givcol, lapack_int *ldgcol, float *givnum,
lapack_int *ldgnum, float *poles, float *difl, float *difr, float *z, lapack_int *k,
float *c, float *s, float *work, lapack_int *iwork, lapack_int *info );

void dlasd6( lapack_int *icompq, lapack_int *nl, lapack_int *nr, lapack_int *sqre,
double *d, double *vf, double *vl, double *alpha, double *beta, lapack_int *idxq,
lapack_int *perm, lapack_int *givptr, lapack_int *givcol, lapack_int *ldgcol, double
```

```
*givnum, lapack_int *ldgnum, double *poles, double *difl, double *difr, double *z,
lapack_int *k, double *c, double *s, double *work, lapack_int *iwork, lapack_int
*info );
```

Include Files

- mkl.h

Description

The routine `?lasd6` computes the *SVD* of an updated upper bidiagonal matrix B obtained by merging two smaller ones by appending a row. This routine is used only for the problem which requires all singular values and optionally singular vector matrices in factored form. B is an n -by- m matrix with $n = n1 + nr + 1$ and $m = n + sqre$. A related subroutine, `?lasd1`, handles the case in which all singular values and singular vectors of the bidiagonal matrix are desired. `?lasd6` computes the *SVD* as follows:

$$B = U(in)* \begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1' & a & Z2' & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix} *VT(in)$$

$= U(out) * (D(out) * VT(out))$

where $Z' = (Z1' \ a \ Z2' \ b) = u' * VT'$, and u is a vector of dimension m with *alpha* and *beta* in the $n1+1$ and $n1+2$ -th entries and zeros elsewhere; and the entry b is empty if $sqre = 0$.

The singular values of B can be computed using $D1$, $D2$, the first components of all the right singular vectors of the lower block, and the last components of all the right singular vectors of the upper block. These components are stored and updated in vf and vl , respectively, in `?lasd6`. Hence U and VT are not explicitly referenced.

The singular values are stored in D . The algorithm consists of two stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or if there is a zero in the Z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `?lasd7`.
2. The second stage consists of calculating the updated singular values. This is done by finding the roots of the secular equation via the routine `?lasd4` (as called by `?lasd8`). This routine also updates vf and vl and computes the distances between the updated singular values and the old singular values. `?lasd6` is called from `?lasda`.

Input Parameters

<i>icompg</i>	Specifies whether singular vectors are to be computed in factored form: = 0: Compute singular values only = 1: Compute singular vectors in factored form as well.
<i>n1</i>	The row dimension of the upper block. $n1 \geq 1$.
<i>nr</i>	The row dimension of the lower block.

	$nr \geq 1$.
<i>sqr</i>	<p>= 0: the lower block is an nr-by-nr square matrix.</p> <p>= 1: the lower block is an nr-by-$(nr+1)$ rectangular matrix.</p> <p>The bidiagonal matrix has row dimension $n=nl+nr+1$, and column dimension $m = n + sqr$.</p>
<i>d</i>	Array, dimension ($nl+nr+1$). On entry $d(1:nl,1:nl)$ contains the singular values of the upper block, and $d(nl+2:n)$ contains the singular values of the lower block.
<i>vf</i>	<p>Array, dimension (m).</p> <p>On entry, $vf(1:nl+1)$ contains the first components of all right singular vectors of the upper block; and $vf(nl+2:m)$ contains the first components of all right singular vectors of the lower block.</p>
<i>vl</i>	<p>Array, dimension (m).</p> <p>On entry, $vl(1:nl+1)$ contains the last components of all right singular vectors of the upper block; and $vl(nl+2:m)$ contains the last components of all right singular vectors of the lower block.</p>
<i>alpha</i>	Contains the diagonal element associated with the added row.
<i>beta</i>	Contains the off-diagonal element associated with the added row.
<i>ldgcol</i>	The leading dimension of the output array <i>givcol</i> , must be at least n .
<i>ldgnum</i>	The leading dimension of the output arrays <i>givnum</i> and <i>poles</i> , must be at least n .
<i>work</i>	Workspace array, dimension ($4m$).
<i>iwork</i>	Workspace array, dimension ($3n$).

Output Parameters

<i>d</i>	On exit $d(1:n)$ contains the singular values of the modified matrix.
<i>vf</i>	On exit, <i>vf</i> contains the first components of all right singular vectors of the bidiagonal matrix.
<i>vl</i>	On exit, <i>vl</i> contains the last components of all right singular vectors of the bidiagonal matrix.
<i>alpha</i>	On exit, the diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(\text{D(I)})), I = 1, n$.
<i>beta</i>	On exit, the off-diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(\text{D(I)})), I = 1, n$.
<i>idxq</i>	Array, dimension (n). This contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, $d(\text{idxq}(i = 1, n))$ will be in ascending order.
<i>perm</i>	Array, dimension (n). The permutations (from deflation and sorting) to be applied to each block. Not referenced if <i>icompg</i> = 0.

<i>givptr</i>	The number of Givens rotations which took place in this subproblem. Not referenced if <i>icompq</i> = 0.
<i>givcol</i>	Array, dimension (<i>ldgcol</i> , 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>givnum</i>	Array, dimension (<i>ldgnum</i> , 2). Each number indicates the C or S value to be used in the corresponding Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>poles</i>	Array, dimension (<i>ldgnum</i> , 2). On exit, <i>poles</i> (1,*) is an array containing the new singular values obtained from solving the secular equation, and <i>poles</i> (2,*) is an array containing the poles in the secular equation. Not referenced if <i>icompq</i> = 0.
<i>difl</i>	Array, dimension (<i>n</i>). On exit, <i>difl</i> (<i>i</i>) is the distance between <i>i</i> -th updated (undeflated) singular value and the <i>i</i> -th (undeflated) old singular value.
<i>difr</i>	Array, dimension (<i>ldgnum</i> , 2) if <i>icompq</i> = 1 and dimension (<i>n</i>) if <i>icompq</i> = 0. On exit, <i>difr</i> (<i>i</i> , 1) is the distance between <i>i</i> -th updated (undeflated) singular value and the <i>i</i> +1-th (undeflated) old singular value. If <i>icompq</i> = 1, <i>difr</i> (1: <i>k</i> , 2) is an array containing the normalizing factors for the right singular vector matrix. See ?lasd8 for details on <i>difl</i> and <i>difr</i> .
<i>z</i>	Array, dimension (<i>m</i>). The first elements of this array contain the components of the deflation-adjusted updating row vector.
<i>k</i>	Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \leq k \leq n$.
<i>c</i>	<i>c</i> contains garbage if <i>sqre</i> = 0 and the C-value of a Givens rotation related to the right null space if <i>sqre</i> = 1.
<i>s</i>	<i>s</i> contains garbage if <i>sqre</i> = 0 and the S-value of a Givens rotation related to the right null space if <i>sqre</i> = 1.
<i>info</i>	= 0: successful exit. < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value. > 0: if <i>info</i> = 1, an singular value did not converge

?lasd7

Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.

Syntax

```
void slasd7( lapack_int *icompq, lapack_int *nl, lapack_int *nr, lapack_int *sqre,
lapack_int *k, float *d, float *z, float *zw, float *vf, float *vfw, float *vl, float
*vlw, float *alpha, float *beta, float *dsigma, lapack_int *idx, lapack_int *idxp,
lapack_int *idxq, lapack_int *perm, lapack_int *givptr, lapack_int *givcol, lapack_int
*ldgcol, float *givnum, lapack_int *ldgnum, float *c, float *s, lapack_int *info );

void dlasd7( lapack_int *icompq, lapack_int *nl, lapack_int *nr, lapack_int *sqre,
lapack_int *k, double *d, double *z, double *zw, double *vf, double *vfw, double *vl,
double *vlw, double *alpha, double *beta, double *dsigma, lapack_int *idx, lapack_int
*idxp, lapack_int *idxq, lapack_int *perm, lapack_int *givptr, lapack_int *givcol,
lapack_int *ldgcol, double *givnum, lapack_int *ldgnum, double *c, double *s,
lapack_int *info );
```

Include Files

- mkl.h

Description

The routine `?lasd7` merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the *Z* vector. For each such occurrence the order of the related secular equation problem is reduced by one. `?lasd7` is called from `?lasd6`.

Input Parameters

<i>icompq</i>	Specifies whether singular vectors are to be computed in compact form, as follows: = 0: Compute singular values only. = 1: Compute singular vectors of upper bidiagonal matrix in compact form.
<i>nl</i>	The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	The row dimension of the lower block. $nr \geq 1$.
<i>sqre</i>	= 0: the lower block is an <i>nr</i> -by- <i>nr</i> square matrix. = 1: the lower block is an <i>nr</i> -by-(<i>nr</i> +1) rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.
<i>d</i>	Array, DIMENSION (<i>n</i>). On entry <i>d</i> contains the singular values of the two submatrices to be combined.
<i>zw</i>	Array, DIMENSION (<i>m</i>). Workspace for <i>z</i> .
<i>vf</i>	Array, DIMENSION (<i>m</i>). On entry, <i>vf</i> (1: <i>nl</i> +1) contains the first components of all right singular vectors of the upper block; and <i>vf</i> (<i>nl</i> +2: <i>m</i>) contains the first components of all right singular vectors of the lower block.

<i>vfw</i>	Array, <code>DIMENSION (m)</code> . Workspace for <i>vf</i> .
<i>vl</i>	Array, <code>DIMENSION (m)</code> . On entry, <i>vl</i> (1: <i>nl</i> +1) contains the last components of all right singular vectors of the upper block; and <i>vl</i> (<i>nl</i> +2: <i>m</i>) contains the last components of all right singular vectors of the lower block.
<i>VLW</i>	Array, <code>DIMENSION (m)</code> . Workspace for <i>VL</i> .
<i>alpha</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlsd7</code> . Contains the diagonal element associated with the added row.
<i>beta</i>	Contains the off-diagonal element associated with the added row.
<i>idx</i>	Workspace array, <code>DIMENSION (n)</code> . This will contain the permutation used to sort the contents of <i>d</i> into ascending order.
<i>idxp</i>	Workspace array, <code>DIMENSION (n)</code> . This will contain the permutation used to place deflated values of <i>d</i> at the end of the array.
<i>idxq</i>	Array, <code>DIMENSION (n)</code> . This contains the permutation which separately sorts the two sub-problems in <i>d</i> into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have <i>nl</i> +1 added to their values.
<i>ldgcol</i>	The leading dimension of the output array <i>givcol</i> , must be at least <i>n</i> .
<i>ldgnum</i>	The leading dimension of the output array <i>givnum</i> , must be at least <i>n</i> .

Output Parameters

<i>k</i>	Contains the dimension of the non-deflated matrix, this is the order of the related secular equation. $1 \leq k \leq n$.
<i>d</i>	On exit, <i>d</i> contains the trailing (<i>n</i> - <i>k</i>) updated singular values (those which were deflated) sorted into increasing order.
<i>z</i>	Array, <code>DIMENSION (m)</code> . On exit, <i>Z</i> contains the updating row vector in the secular equation.
<i>vf</i>	On exit, <i>vf</i> contains the first components of all right singular vectors of the bidiagonal matrix.
<i>vl</i>	On exit, <i>vl</i> contains the last components of all right singular vectors of the bidiagonal matrix.
<i>dsigma</i>	Array, <code>DIMENSION (n)</code> . Contains a copy of the diagonal elements (<i>k</i> -1 singular values and one zero) in the secular equation.

<i>idxp</i>	On output, <i>idxp</i> (2: <i>k</i>) points to the nondeflated <i>d</i> -values and <i>idxp</i> (<i>k</i> +1: <i>n</i>) points to the deflated singular values.
<i>perm</i>	Array, DIMENSION (<i>n</i>). The permutations (from deflation and sorting) to be applied to each singular block. Not referenced if <i>icompq</i> = 0.
<i>givptr</i>	The number of Givens rotations which took place in this subproblem. Not referenced if <i>icompq</i> = 0.
<i>givcol</i>	Array, DIMENSION (<i>ldgcol</i> , 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>givnum</i>	Array, DIMENSION (<i>ldgnum</i> , 2). Each number indicates the C or S value to be used in the corresponding Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>c</i>	If <i>sqr</i> = 0, then <i>c</i> contains garbage, and if <i>sqr</i> = 1, then <i>c</i> contains C-value of a Givens rotation related to the right null space.
<i>s</i>	If <i>sqr</i> = 0, then <i>s</i> contains garbage, and if <i>sqr</i> = 1, then <i>s</i> contains S-value of a Givens rotation related to the right null space.
<i>info</i>	= 0: successful exit. < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value.

?lasd8

Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.

Syntax

```
void slasd8( lapack_int *icompq, lapack_int *k, float *d, float *z, float *vf, float
*vl, float *difl, float *difr, lapack_int *lddifr, float *dsigma, float *work,
lapack_int *info );

void dlasd8( lapack_int *icompq, lapack_int *k, double *d, double *z, double *vf, double
*vl, double *difl, double *difr, lapack_int *lddifr, double *dsigma, double *work,
lapack_int *info );
```

Include Files

- mkl.h

Description

The routine ?lasd8 finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to ?lasd4, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd8 is called from ?lasd6.

Input Parameters

<i>icompq</i>	Specifies whether singular vectors are to be computed in factored form in the calling routine: = 0: Compute singular values only. = 1: Compute singular vectors in factored form as well.
<i>k</i>	The number of terms in the rational function to be solved by ?lasd4. $k \geq 1$.
<i>z</i>	Array, DIMENSION (<i>k</i>). The first <i>k</i> elements of this array contain the components of the deflation-adjusted updating row vector.
<i>vf</i>	Array, DIMENSION (<i>k</i>). On entry, <i>vf</i> contains information passed through dbede8.
<i>vl</i>	Array, DIMENSION (<i>k</i>). On entry, <i>vl</i> contains information passed through dbede8.
<i>lddifr</i>	The leading dimension of the output array <i>difr</i> , must be at least <i>k</i> .
<i>dsigma</i>	Array, DIMENSION (<i>k</i>). The first <i>k</i> elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.
<i>work</i>	Workspace array, DIMENSION at least (3 <i>k</i>).

Output Parameters

<i>d</i>	Array, DIMENSION (<i>k</i>). On output, <i>D</i> contains the updated singular values.
<i>z</i>	Updated on exit.
<i>vf</i>	On exit, <i>vf</i> contains the first <i>k</i> components of the first components of all right singular vectors of the bidiagonal matrix.
<i>vl</i>	On exit, <i>vl</i> contains the first <i>k</i> components of the last components of all right singular vectors of the bidiagonal matrix.
<i>difl</i>	Array, DIMENSION (<i>k</i>). On exit, $difl(i) = d(i) - dsigma(i)$.
<i>difr</i>	Array, DIMENSION (<i>lddifr</i> , 2) if <i>icompq</i> = 1 and DIMENSION (<i>k</i>) if <i>icompq</i> = 0. On exit, $difr(i,1) = d(i) - dsigma(i+1)$, $difr(k,1)$ is not defined and will not be referenced. If <i>icompq</i> = 1, $difr(1:k,2)$ is an array containing the normalizing factors for the right singular vector matrix.
<i>dsigma</i>	The elements of this array may be very slightly altered in value.
<i>info</i>	= 0: successful exit. < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value.

> 0: If *info* = 1, an singular value did not converge.

?lasd9

*Finds the square roots of the roots of the secular equation, and stores, for each element in *D*, the distance to its two nearest poles. Used by ?bdsdc.*

Syntax

```
void slasd9( lapack_int *icompq, lapack_int *k, float *d, float *z, float *vf, float
*vl, float *difl, float *difr, float *dsigma, float *work, lapack_int *info );

void dlasd9( lapack_int *icompq, lapack_int *k, double *d, double *z, double *vf, double
*vl, double *difl, double *difr, double *dsigma, double *work, lapack_int *info );
```

Include Files

- mkl.h

Description

The routine ?lasd9 finds the square roots of the roots of the secular equation, as defined by the values in *dsigma* and *z*. It makes the appropriate calls to ?lasd4, and stores, for each element in *d*, the distance to its two nearest poles (elements in *dsigma*). It also updates the arrays *vf* and *vl*, the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd9 is called from ?lasd7.

Input Parameters

<i>icompq</i>	Specifies whether singular vectors are to be computed in factored form in the calling routine: If <i>icompq</i> = 0, compute singular values only; If <i>icompq</i> = 1, compute singular vector matrices in factored form also.
<i>k</i>	The number of terms in the rational function to be solved by slasd4. $k \geq 1$.
<i>dsigma</i>	Array, DIMENSION(<i>k</i>). The first <i>k</i> elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.
<i>z</i>	Array, DIMENSION (<i>k</i>). The first <i>k</i> elements of this array contain the components of the deflation-adjusted updating row vector.
<i>vf</i>	Array, DIMENSION(<i>k</i>). On entry, <i>vf</i> contains information passed through sbded8.
<i>vl</i>	Array, DIMENSION(<i>k</i>). On entry, <i>vl</i> contains information passed through sbded8.
<i>work</i>	Workspace array, DIMENSION at least (3 <i>k</i>).

Output Parameters

<i>d</i>	Array, DIMENSION(<i>k</i>). <i>d</i> (<i>i</i>) contains the updated singular values.
----------	---

<code>vf</code>	On exit, <code>vf</code> contains the first k components of the first components of all right singular vectors of the bidiagonal matrix.
<code>vl</code>	On exit, <code>vl</code> contains the first k components of the last components of all right singular vectors of the bidiagonal matrix.
<code>difl</code>	Array, $\text{DIMENSION}(k)$. On exit, $difl(i) = d(i) - dsigma(i)$.
<code>difr</code>	Array, $\text{DIMENSION}(ldu, 2)$ if $icompq = 1$ and $\text{DIMENSION}(k)$ if $icompq = 0$. On exit, $difr(i, 1) = d(i) - dsigma(i+1)$, $difr(k, 1)$ is not defined and will not be referenced. If $icompq = 1$, $difr(1:k, 2)$ is an array containing the normalizing factors for the right singular vector matrix.
<code>info</code>	$= 0$: successful exit. < 0 : if $info = -i$, the i -th argument had an illegal value. > 0 : If $info = 1$, an singular value did not converge

?lasda

Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc.

Syntax

```
void slasda( lapack_int *icompq, lapack_int *smlsiz, lapack_int *n, lapack_int *sqre,
float *d, float *e, float *u, lapack_int *ldu, float *vt, lapack_int *k, float *difl,
float *difr, float *z, float *poles, lapack_int *givptr, lapack_int *givcol, lapack_int
*ldgcol, lapack_int *perm, float *givnum, float *c, float *s, float *work, lapack_int
*iwork, lapack_int *info );
```

```
void dlasda( lapack_int *icompq, lapack_int *smlsiz, lapack_int *n, lapack_int *sqre,
double *d, double *e, double *u, lapack_int *ldu, double *vt, lapack_int *k, double
*difl, double *difr, double *z, double *poles, lapack_int *givptr, lapack_int *givcol,
lapack_int *ldgcol, lapack_int *perm, double *givnum, double *c, double *s, double
*work, lapack_int *iwork, lapack_int *info );
```

Include Files

- mkl.h

Description

Using a divide and conquer approach, ?lasda computes the singular value decomposition (SVD) of a real upper bidiagonal n -by- m matrix B with diagonal d and off-diagonal e , where $m = n + sqre$.

The algorithm computes the singular values in the $SVDB = U^*S^*VT$. The orthogonal matrices U and VT are optionally computed in compact form. A related subroutine ?lasd0 computes the singular values and the singular vectors in explicit form.

Input Parameters

<i>icompq</i>	Specifies whether singular vectors are to be computed in compact form, as follows: = 0: Compute singular values only. = 1: Compute singular vectors of upper bidiagonal matrix in compact form.
<i>smlsiz</i>	The maximum size of the subproblems at the bottom of the computation tree.
<i>n</i>	The row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array <i>d</i> .
<i>sqre</i>	Specifies the column dimension of the bidiagonal matrix. If <i>sqre</i> = 0: the bidiagonal matrix has column dimension $m = n$ If <i>sqre</i> = 1: the bidiagonal matrix has column dimension $m = n + 1$.
<i>d</i>	Array, DIMENSION (<i>n</i>). On entry, <i>d</i> contains the main diagonal of the bidiagonal matrix.
<i>e</i>	Array, DIMENSION (<i>m</i> - 1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, <i>e</i> is destroyed.
<i>ldu</i>	The leading dimension of arrays <i>u</i> , <i>vt</i> , <i>difl</i> , <i>difr</i> , <i>poles</i> , <i>givnum</i> , and <i>z</i> . $ldu \geq n$.
<i>ldgcol</i>	The leading dimension of arrays <i>givcol</i> and <i>perm</i> . $ldgcol \geq n$.
<i>work</i>	Workspace array, DIMENSION ($6n + (smlsiz + 1)^2$).
<i>iwork</i>	Workspace array, <i>Dimension</i> must be at least (7 <i>n</i>).

Output Parameters

<i>d</i>	On exit <i>d</i> , if <i>info</i> = 0, contains the singular values of the bidiagonal matrix.
<i>u</i>	Array, DIMENSION (<i>ldu</i> , <i>smlsiz</i>) if <i>icompq</i> = 1. Not referenced if <i>icompq</i> = 0. If <i>icompq</i> = 1, on exit, <i>u</i> contains the left singular vector matrices of all subproblems at the bottom level.
<i>vt</i>	Array, DIMENSION (<i>ldu</i> , <i>smlsiz</i> + 1) if <i>icompq</i> = 1, and not referenced if <i>icompq</i> = 0. If <i>icompq</i> = 1, on exit, <i>vt</i> ' contains the right singular vector matrices of all subproblems at the bottom level.
<i>k</i>	Array, DIMENSION (<i>n</i>) if <i>icompq</i> = 1 and DIMENSION (1) if <i>icompq</i> = 0. If <i>icompq</i> = 1, on exit, <i>k</i> (<i>i</i>) is the dimension of the <i>i</i> -th secular equation on the computation tree.
<i>difl</i>	REAL for slasda DOUBLE PRECISION for dlasda.

Array, DIMENSION (*ldu*, *nlvl*),

where $nlvl = \text{floor}(\log_2(n/smlsiz))$.

difr

Array,

DIMENSION (*ldu*, $2\ nlvl$) if *icompq* = 1 and

DIMENSION (*n*) if *icompq* = 0.

If *icompq* = 1, on exit, *difr*(1:*n*, *i*) and *difr*(1:*n*, $2i-1$) record distances between singular values on the *i*-th level and singular values on the (*i*-1)-th level, and *difr*(1:*n*, $2i$) contains the normalizing factors for the right singular vector matrix. See ?lasd8 for details.

z

Array,

DIMENSION (*ldu*, *nlvl*) if *icompq* = 1 and

DIMENSION (*n*) if *icompq* = 0. The first *k* elements of *z*(1, *i*) contain the components of the deflation-adjusted updating row vector for subproblems on the *i*-th level.

poles

Array, DIMENSION(*ldu*, $2*nlvl$)

if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *poles*(1, $2i-1$) and *poles*(1, $2i$) contain the new and old singular values involved in the secular equations on the *i*-th level.

givptr

Array, DIMENSION (*n*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *givptr*(*i*) records the number of Givens rotations performed on the *i*-th problem on the computation tree.

givcol

Array, DIMENSION(*ldgcol*, $2*nlvl$) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givcol*(1, $2i-1$) and *givcol*(1, $2i$) record the locations of Givens rotations performed on the *i*-th level on the computation tree.

perm

Array, DIMENSION (*ldgcol*, *nlvl*) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, *perm* (1, *i*) records permutations done on the *i*-th level of the computation tree.

givnum

Array DIMENSION (*ldu*, $2*nlvl$) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givnum*(1, $2i-1$) and *givnum*(1, $2i$) record the C- and S-values of Givens rotations performed on the *i*-th level on the computation tree.

c

Array,

DIMENSION (*n*) if *icompq* = 1, and

DIMENSION (1) if *icompq* = 0.

If *icompq* = 1 and the *i*-th subproblem is not square, on exit, *c*(*i*) contains the C-value of a Givens rotation related to the right null space of the *i*-th subproblem.

s

Array,

DIMENSION (*n*) if *icompq* = 1, and

DIMENSION (1) if *icompq* = 0.

If $icompg = 1$ and the i -th subproblem is not square, on exit, $s(i)$ contains the S -value of a Givens rotation related to the right null space of the i -th subproblem.

info

= 0: successful exit.
 < 0: if *info* = $-i$, the i -th argument had an illegal value
 > 0: If *info* = 1, an singular value did not converge

?lasdq

Computes the SVD of a real bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc.

Syntax

```
void slasdq( char *uplo, lapack_int *sqre, lapack_int *n, lapack_int *ncvt, lapack_int
*nru, lapack_int *ncc, float *d, float *e, float *vt, lapack_int *ldvt, float *u,
lapack_int *ldu, float *c, lapack_int *ldc, float *work, lapack_int *info );

void dlasdq( char *uplo, lapack_int *sqre, lapack_int *n, lapack_int *ncvt, lapack_int
*nru, lapack_int *ncc, double *d, double *e, double *vt, lapack_int *ldvt, double *u,
lapack_int *ldu, double *c, lapack_int *ldc, double *work, lapack_int *info );
```

Include Files

- mkl.h

Description

The routine ?lasdq computes the singular value decomposition (SVD) of a real (upper or lower) bidiagonal matrix with diagonal d and off-diagonal e , accumulating the transformations if desired. If B is the input bidiagonal matrix, the algorithm computes orthogonal matrices Q and P such that $B = Q^T S P^T$. The singular values S are overwritten on d .

The input matrix U is changed to U^*Q if desired.

The input matrix VT is changed to $P^T * VT$ if desired.

The input matrix C is changed to $Q^T * C$ if desired.

Input Parameters

uplo

On entry, *uplo* specifies whether the input bidiagonal matrix is upper or lower bidiagonal.

If *uplo* = 'U' or 'u', B is upper bidiagonal;

If *uplo* = 'L' or 'l', B is lower bidiagonal.

sqre

= 0: then the input matrix is n -by- n .

= 1: then the input matrix is n -by- $(n+1)$ if *uplu* = 'U' and $(n+1)$ -by- n if *uplu*

= 'L'. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.

<i>n</i>	On entry, <i>n</i> specifies the number of rows and columns in the matrix. <i>n</i> must be at least 0.
<i>ncvt</i>	On entry, <i>ncvt</i> specifies the number of columns of the matrix <i>VT</i> . <i>ncvt</i> must be at least 0.
<i>nru</i>	On entry, <i>nru</i> specifies the number of rows of the matrix <i>U</i> . <i>nru</i> must be at least 0.
<i>ncc</i>	On entry, <i>ncc</i> specifies the number of columns of the matrix <i>C</i> . <i>ncc</i> must be at least 0.
<i>d</i>	Array, <code>DIMENSION (n)</code> . On entry, <i>d</i> contains the diagonal entries of the bidiagonal matrix.
<i>e</i>	Array, <code>DIMENSION</code> is $(n-1)$ if <i>sGRE</i> = 0 and <i>n</i> if <i>sGRE</i> = 1. On entry, the entries of <i>e</i> contain the off-diagonal entries of the bidiagonal matrix.
<i>vt</i>	Array, <code>DIMENSION (ldvt, ncvt)</code> . On entry, contains a matrix which on exit has been premultiplied by P^T , dimension <i>n</i> -by- <i>ncvt</i> if <i>sGRE</i> = 0 and $(n+1)$ -by- <i>ncvt</i> if <i>sGRE</i> = 1 (not referenced if <i>ncvt</i> =0).
<i>ldvt</i>	On entry, <i>ldvt</i> specifies the leading dimension of <i>vt</i> as declared in the calling (sub) program. <i>ldvt</i> must be at least 1. If <i>ncvt</i> is nonzero, <i>ldvt</i> must also be at least <i>n</i> .
<i>u</i>	Array, <code>DIMENSION (ldu, n)</code> . On entry, contains a matrix which on exit has been postmultiplied by <i>Q</i> , dimension <i>nru</i> -by- <i>n</i> if <i>sGRE</i> = 0 and <i>nru</i> -by- $(n+1)$ if <i>sGRE</i> = 1 (not referenced if <i>nru</i> =0).
<i>ldu</i>	On entry, <i>ldu</i> specifies the leading dimension of <i>u</i> as declared in the calling (sub) program. <i>ldu</i> must be at least $\max(1, nru)$.
<i>c</i>	Array, <code>DIMENSION (ldc, ncc)</code> . On entry, contains an <i>n</i> -by- <i>ncc</i> matrix which on exit has been premultiplied by Q' , dimension <i>n</i> -by- <i>ncc</i> if <i>sGRE</i> = 0 and $(n+1)$ -by- <i>ncc</i> if <i>sGRE</i> = 1 (not referenced if <i>ncc</i> =0).
<i>ldc</i>	On entry, <i>ldc</i> specifies the leading dimension of <i>C</i> as declared in the calling (sub) program. <i>ldc</i> must be at least 1. If <i>ncc</i> is non-zero, <i>ldc</i> must also be at least <i>n</i> .
<i>work</i>	Array, <code>DIMENSION (4n)</code> . This is a workspace array. Only referenced if one of <i>ncvt</i> , <i>nru</i> , or <i>ncc</i> is nonzero, and if <i>n</i> is at least 2.

Output Parameters

<i>d</i>	On normal exit, <i>d</i> contains the singular values in ascending order.
<i>e</i>	On normal exit, <i>e</i> will contain 0. If the algorithm does not converge, <i>d</i> and <i>e</i> will contain the diagonal and superdiagonal entries of a bidiagonal matrix orthogonally equivalent to the one given as input.
<i>vt</i>	On exit, the matrix has been premultiplied by P' .
<i>u</i>	On exit, the matrix has been postmultiplied by <i>Q</i> .
<i>c</i>	On exit, the matrix has been premultiplied by Q' .

info On exit, a value of 0 indicates a successful exit. If *info* < 0, argument number *-info* is illegal. If *info* > 0, the algorithm did not converge, and *info* specifies how many superdiagonals did not converge.

?lasdt

Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.

Syntax

```
void slasdt( lapack_int *n, lapack_int *lvl, lapack_int *nd, lapack_int *inode,
lapack_int *ndiml, lapack_int *ndimr, lapack_int *msub );

void dlasdt( lapack_int *n, lapack_int *lvl, lapack_int *nd, lapack_int *inode,
lapack_int *ndiml, lapack_int *ndimr, lapack_int *msub );
```

Include Files

- mkl.h

Description

The routine creates a tree of subproblems for bidiagonal divide and conquer.

Input Parameters

n On entry, the number of diagonal elements of the bidiagonal matrix.

msub On entry, the maximum row dimension each subproblem at the bottom of the tree can be of.

Output Parameters

lvl On exit, the number of levels on the computation tree.

nd On exit, the number of nodes on the tree.

inode Array, DIMENSION (*n*). On exit, centers of subproblems.

ndiml Array, DIMENSION (*n*). On exit, row dimensions of left children.

ndimr Array, DIMENSION (*n*). On exit, row dimensions of right children.

?laset

Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.

Syntax

```
lapack_int LAPACKE_slaset (int matrix_layout , char uplo , lapack_int m , lapack_int
n , float alpha , float beta , float * a , lapack_int lda );

lapack_int LAPACKE_dlaset (int matrix_layout , char uplo , lapack_int m , lapack_int
n , double alpha , double beta , double * a , lapack_int lda );
```

```
lapack_int LAPACKE_claset (int matrix_layout , char uplo , lapack_int m , lapack_int
n , lapack_complex_float alpha , lapack_complex_float beta , lapack_complex_float * a ,
lapack_int lda );

lapack_int LAPACKE_zlaset (int matrix_layout , char uplo , lapack_int m , lapack_int
n , lapack_complex_double alpha , lapack_complex_double beta , lapack_complex_double *
a , lapack_int lda );
```

Include Files

- mkl.h

Description

The routine initializes an m -by- n matrix A to β on the diagonal and α on the off-diagonals.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies the part of the matrix A to be set. If <i>uplo</i> = 'U', upper triangular part is set; the strictly lower triangular part of A is not changed. If <i>uplo</i> = 'L': lower triangular part is set; the strictly upper triangular part of A is not changed. Otherwise: All of the matrix A is set.
<i>m</i>	The number of rows of the matrix A . $m \geq 0$.
<i>n</i>	The number of columns of the matrix A . $n \geq 0$.
<i>alpha, beta</i>	The constants to which the off-diagonal and diagonal elements are to be set, respectively.
<i>a</i>	Array, size at least $\max(1, lda \cdot n)$ for column major and $\max(1, lda \cdot m)$ for row major layout. The array <i>a</i> contains the m -by- n matrix A .
<i>lda</i>	The leading dimension of the array <i>a</i> . $lda \geq \max(1, m)$ for column major layout and $lda \geq \max(1, n)$ for row major layout.

Output Parameters

<i>a</i>	On exit, the leading m -by- n submatrix of A is set as follows: if <i>uplo</i> = 'U', $A_{ij} = \alpha, 1 \leq i \leq j-1, 1 \leq j \leq n$, if <i>uplo</i> = 'L', $A_{ij} = \alpha, j+1 \leq i \leq m, 1 \leq j \leq n$, otherwise, $A_{ij} = \alpha, 1 \leq i \leq m, 1 \leq j \leq n, i \neq j$,
----------	---

and, for all *uplo*, $A_{ii} = \text{beta}$, $1 \leq i \leq \min(m, n)$.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = *i* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?lasrt

Sorts numbers in increasing or decreasing order.

Syntax

```
lapack_int LAPACKE_slasrt (char id , lapack_int n , float * d );
lapack_int LAPACKE_dlasrt (char id , lapack_int n , double * d );
```

Include Files

- mkl.h

Description

The routine ?lasrt sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20 . Dimension of *stack* limits *n* to about 2^{32} .

Input Parameters

A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

id = 'I': sort *d* in increasing order;
 = 'D': sort *d* in decreasing order.

n The length of the array *d*.

d On entry, the array to be sorted.

Output Parameters

d On exit, *d* has been sorted into increasing order
 ($d[0] \leq d[1] \leq \dots \leq d[n-1]$) or into decreasing order
 ($d[0] \geq d[1] \geq \dots \geq d[n-1]$), depending on *id*.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```

lapack_int LAPACKE_slaswp (int matrix_layout , lapack_int n , float * a , lapack_int
lda , lapack_int k1 , lapack_int k2 , const lapack_int * ipiv , lapack_int incx );

lapack_int LAPACKE_dlaswp (int matrix_layout , lapack_int n , double * a , lapack_int
lda , lapack_int k1 , lapack_int k2 , const lapack_int * ipiv , lapack_int incx );

lapack_int LAPACKE_claswp (int matrix_layout , lapack_int n , lapack_complex_float *
a , lapack_int lda , lapack_int k1 , lapack_int k2 , const lapack_int * ipiv ,
lapack_int incx );

lapack_int LAPACKE_zlaswp (int matrix_layout , lapack_int n , lapack_complex_double *
a , lapack_int lda , lapack_int k1 , lapack_int k2 , const lapack_int * ipiv ,
lapack_int incx );

```

Include Files

- mkl.h

Description

The routine performs a series of row interchanges on the matrix *A*. One row interchange is initiated for each of rows *k1* through *k2* of *A*.

Input Parameters

A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>n</i>	The number of columns of the matrix <i>A</i> .
<i>a</i>	Array, size $\max(1, lda*n)$ for column major and $\max(1, lda*mm)$ for row major layout. Here <i>mm</i> is not less than maximum of values $ipiv[k1-1+j* incx]$, $0 \leq j < k2-k1$. Array <i>a</i> contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> .
<i>lda</i>	The leading dimension of the array <i>a</i> .
<i>k1</i>	The first element of <i>ipiv</i> for which a row interchange will be done.
<i>k2</i>	The last element of <i>ipiv</i> for which a row interchange will be done.
<i>ipiv</i>	Array, size $k1 + (k2-k1) * incx $. The vector of pivot indices. Only the elements in positions <i>k1</i> through <i>k2</i> of <i>ipiv</i> are accessed. $ipiv(k) = l$ implies rows <i>k</i> and <i>l</i> are to be interchanged.

incx The increment between successive values of *ipiv*. If *ipiv* is negative, the pivots are applied in reverse order.

Output Parameters

a On exit, the permuted matrix.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?latm1

Computes the entries of a matrix as specified.

Syntax

```
void slatm1 (lapack_int *mode, *cond, lapack_int *irsign, lapack_int *idist, lapack_int *iseed, float *d, lapack_int *n, lapack_int *info);
```

```
void dlatm1 (lapack_int *mode, *cond, lapack_int *irsign, lapack_int *idist, lapack_int *iseed, double *d, lapack_int *n, lapack_int *info);
```

```
void clatm1 (lapack_int *mode, *cond, lapack_int *irsign, lapack_int *idist, lapack_int *iseed, lapack_complex *d, lapack_int *n, lapack_int *info);
```

```
void zlatm1 (lapack_int *mode, *cond, lapack_int *irsign, lapack_int *idist, lapack_int *iseed, lapack_complex_double *d, lapack_int *n, lapack_int *info);
```

Include Files

- mkl.h

Description

The ?latm1 routine computes the entries of $D(1..n)$ as specified by *mode*, *cond* and *irsign*. *idist* and *iseed* determine the generation of random numbers.

?latm1 is called by slatmr (for slatm1 and dlatm1), and by clatmr (for clatm1 and zlatm1) to generate random test matrices for LAPACK programs.

Input Parameters

mode On entry describes how *d* is to be computed:

mode = 0 means do not change *d*.

mode = 1 sets $d[0] = 1$ and $d[1:n - 1] = 1.0/cond$

mode = 2 sets $d[0:n - 2] = 1$ and $d[n - 1] = 1.0/cond$

mode = 3 sets $d[i - 1] = cond^{-(i-1)/(n-1)}$

mode = 4 sets $d[i - 1] = 1 - (i-1)/(n-1) * (1 - 1/cond)$

$mode = 5$ sets d to random numbers in the range $(1/cond, 1)$ such that their logarithms are uniformly distributed.

$mode = 6$ sets d to random numbers from same distribution as the rest of the matrix.

$mode < 0$ has the same meaning as $abs(mode)$, except that the order of the elements of d is reversed.

Thus if $mode$ is positive, d has entries ranging from 1 to $1/cond$, if negative, from $1/cond$ to 1.

cond On entry, used as described under *mode* above. If used, it must be ≥ 1 .

irsign On entry, if *mode* is not -6, 0, or 6, determines sign of entries of d .
If *irsign* = 0, entries of d are unchanged.
If *irsign* = 1, each entry of d is multiplied by a random complex number uniformly distributed with absolute value 1.

idist Specifies the distribution of the random numbers.
For *slatm1* and *dlatm1*:
= 1: uniform (0,1)
= 2: uniform (-1,1)
= 3: normal (0,1)
For *clatm1* and *zlatm1*:
= 1: real and imaginary parts each uniform (0,1)
= 2: real and imaginary parts each uniform (-1,1)
= 3: real and imaginary parts each normal (0,1)
= 4: complex number uniform in disk(0, 1)

iseed Array, size (4).
Specifies the seed of the random number generator. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers. The values of *iseed*[3] are changed on exit, and can be used in the next call to *?latm1* to continue the same random number sequence.

d Array, size n .

n Number of entries of d .

Output Parameters

iseed On exit, the seed is updated.

d On exit, d is updated, unless *mode* = 0.

info If *info* = 0, the execution is successful.

If *info* = -1, *mode* is not in range -6 to 6.

If *info* = -2, *mode* is neither -6, 0 nor 6, and *irsign* is neither 0 nor 1.

If `info = -3`, `mode` is neither -6, 0 nor 6 and `cond` is less than 1.
 If `info = -4`, `mode` equals 6 or -6 and `idist` is not in range 1 to 4.
 If `info = -7`, `n` is negative.

?latm2

Returns an entry of a random matrix.

Syntax

```
float slatm2 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int
*kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, float *d, lapack_int *igrade,
float *dl, float *dr, lapack_int *ipvtng, lapack_int *iwork, float *sparse);

double dlatm2 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int
*kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, double *d, lapack_int
*igrade, double *dl, double *dr, lapack_int *ipvtng, lapack_int *iwork, double *sparse);
```

The data types for complex variations depend on whether or not the application links with Gnu Fortran (gfortran) libraries.

For non-gfortran (libmkl_intel_*) interface libraries:

```
void clatm2 (lapack_complex_float *res, lapack_int *m, lapack_int *n, lapack_int *i,
lapack_int *j, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed,
lapack_complex_float *d, lapack_int *igrade, lapack_complex_float *dl,
lapack_complex_float *dr, lapack_int *ipvtng, lapack_int *iwork, float *sparse);

void zlatm2 (lapack_complex_double *res, lapack_int *m, lapack_int *n, lapack_int *i,
lapack_int *j, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed,
lapack_complex_double *d, lapack_int *igrade, lapack_complex_double *dl,
lapack_complex_double *dr, lapack_int *ipvtng, lapack_int *iwork, double *sparse);
```

For gfortran (libmkl_gf_*) interface libraries:

```
lapack_complex_float clatm2 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int
*j, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed,
lapack_complex_float *d, lapack_int *igrade, lapack_complex_float *dl,
lapack_complex_float *dr, lapack_int *ipvtng, lapack_int *iwork, float *sparse);

lapack_complex_double zlatm2 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int
*j, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed,
lapack_complex_double *d, lapack_int *igrade, lapack_complex_double *dl,
lapack_complex_double *dr, lapack_int *ipvtng, lapack_int *iwork, double *sparse);
```

To understand the difference between the non-gfortran and gfortran interfaces and when to use each of them, see Dynamic Libraries in the lib/intel64 Directory in the *oneAPI Math Kernel Library Developer Guide*.

Include Files

- mkl.h

Description

The ?latm2 routine returns entry (*i*, *j*) of a random matrix of dimension (*m*, *n*). It is called by the ?latmr routine in order to build random test matrices. No error checking on parameters is done, because this routine is called in a tight loop by ?latmr which has already checked the parameters.

Use of `?latm2` differs from `?latm3` in the order in which the random number generator is called to fill in random matrix entries. With `?latm2`, the generator is called to fill in the pivoted matrix columnwise. With `?latm2`, the generator is called to fill in the matrix columnwise, after which it is pivoted. Thus, `?latm3` can be used to construct random matrices which differ only in their order of rows and/or columns. `?latm2` is used to construct band matrices while avoiding calling the random number generator for entries outside the band (and therefore generating random numbers).

The matrix whose (i, j) entry is returned is constructed as follows (this routine only computes one entry):

- If i is outside $(1..m)$ or j is outside $(1..n)$, returns zero (this is convenient for generating matrices in band format).
- Generate a matrix A with random entries of distribution *idist*.
- Set the diagonal to D .
- Grade the matrix, if desired, from the left (by dl) and/or from the right (by dr or dl) as specified by *igrade*.
- Permute, if desired, the rows and/or columns as specified by *ipvtng* and *iwork*.
- Band the matrix to have lower bandwidth kl and upper bandwidth ku .
- Set random entries to zero as specified by *sparse*.

Input Parameters

<i>m</i>	Number of rows of the matrix.
<i>n</i>	Number of columns of the matrix.
<i>i</i>	Row of the entry to be returned.
<i>j</i>	Column of the entry to be returned.
<i>kl</i>	Lower bandwidth.
<i>ku</i>	Upper bandwidth.
<i>idist</i>	On entry, <i>idist</i> specifies the type of distribution to be used to generate a random matrix . for <code>slatm2</code> and <code>dlatm2</code> : = 1: uniform (0,1) = 2: uniform (-1,1) = 3: normal (0,1) for <code>clatm2</code> and <code>zlatm2</code> : = 1: real and imaginary parts each uniform (0,1) = 2: real and imaginary parts each uniform (-1,1) = 3: real and imaginary parts each normal (0,1) = 4: complex number uniform in disk (0, 1)
<i>iseed</i>	Array, size 4. Seed for the random number generator.
<i>d</i>	Array, size $(\min(i, j))$. Diagonal entries of matrix.
<i>igrade</i>	Specifies grading of matrix as follows: = 0: no grading = 1: matrix premultiplied by $\text{diag}(dl)$

= 2: matrix postmultiplied by `diag(dr)`

= 3: matrix premultiplied by `diag(dl)` and postmultiplied by `diag(dr)`

= 4: matrix premultiplied by `diag(dl)` and postmultiplied by `inv(diag(dl))`

For `slatm2` and `slatm2`:

= 5: matrix premultiplied by `diag(dl)` and postmultiplied by `diag(dl)`

For `clatm2` and `zlatm2`:

= 5: matrix premultiplied by `diag(dl)` and postmultiplied by `diag(conjg(dl))`

= 6: matrix premultiplied by `diag(dl)` and postmultiplied by `diag(dl)`

dl Array, size (*i* or *j*), as appropriate.
Left scale factors for grading matrix.

dr Array, size (*i* or *j*), as appropriate.
Right scale factors for grading matrix.

ipvtng On entry specifies pivoting permutations as follows:
= 0: none
= 1: row pivoting
= 2: column pivoting
= 3: full pivoting, i.e., on both sides

iwork Array, size (*i* or *j*), as appropriate. This array specifies the permutation used. The row (or column) in position *k* was originally in position `iwork[k - 1]`. This differs from *iwork* for `?latm3`.

sparse Specifies the sparsity of the matrix. If sparse matrix is to be generated, *sparse* should lie between 0 and 1. A uniform (0, 1) random number *x* is generated and compared to *sparse*. If *x* is larger the matrix entry is unchanged and if *x* is smaller the entry is set to zero. Thus on the average a fraction *sparse* of the entries will be set to zero.

Output Parameters

iseed On exit, the seed is updated.

Return Values

The function returns an entry of a random matrix (for complex variations `libmkl_gf_*` interface layer/ libraries return the result as the parameter *res*).

?latm3

Returns set entry of a random matrix.

Syntax

```
float slatm3 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int
*isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int
*iseed, float *d, lapack_int *igrade, float *dl, float *dr, lapack_int *ipvtng,
lapack_int *iwork, float *sparse);
```

```
double dlatm3 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int
*isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int
*iseed, double *d, lapack_int *igrade, double *dl, double *dr, lapack_int *ipvtng,
lapack_int *iwork, double *sparse);
```

The data types for complex variations depend on whether or not the application links with Gnu Fortran (gfortran) libraries.

For non-gfortran (libmkl_intel_*) interface libraries:

```
void clatm3 (lapack_complex_float *res, lapack_int *m, lapack_int *n, lapack_int *i,
lapack_int *j, lapack_int *isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku,
lapack_int *idist, lapack_int *iseed, lapack_complex_float *d, lapack_int *igrade,
lapack_complex_float *dl, lapack_complex_float *dr, lapack_int *ipvtng, lapack_int
*iwork, float *sparse);
```

```
void zlatm3 (lapack_complex_double *res, lapack_int *m, lapack_int *n, lapack_int *i,
lapack_int *j, lapack_int *isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku,
lapack_int *idist, lapack_int *iseed, lapack_complex_double *d, lapack_int *igrade,
lapack_complex_double *dl, lapack_complex_double *dr, lapack_int *ipvtng, lapack_int
*iwork, double *sparse);
```

For gfortran (libmkl_gf_*) interface libraries:

```
lapack_complex_float clatm3 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int
*j, lapack_int *isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku, lapack_int
*idist, lapack_int *iseed, lapack_complex_float *d, lapack_int *igrade,
lapack_complex_float *dl, lapack_complex_float *dr, lapack_int *ipvtng, lapack_int
*iwork, float *sparse);
```

```
lapack_complex_double zlatm3 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int
*j, lapack_int *isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku, lapack_int
*idist, lapack_int *iseed, lapack_complex_double *d, lapack_int *igrade,
lapack_complex_double *dl, lapack_complex_double *dr, lapack_int *ipvtng, lapack_int
*iwork, double *sparse);
```

To understand the difference between the non-gfortran and gfortran interfaces and when to use each of them, see Dynamic Libraries in the lib/intel64 Directory in the *oneAPI Math Kernel Library Developer Guide*.

Include Files

- mkl.h

Description

The ?latm3 routine returns the (*isub*, *jsub*) entry of a random matrix of dimension (*m*, *n*) described by the other parameters. (*isub*, *jsub*) is the final position of the (*i*, *j*) entry after pivoting according to *ipvtng* and *iwork*. ?latm3 is called by the ?latmr routine in order to build random test matrices. No error checking on parameters is done, because this routine is called in a tight loop by ?latmr which has already checked the parameters.

Use of `?latm3` differs from `?latm2` in the order in which the random number generator is called to fill in random matrix entries. With `?latm2`, the generator is called to fill in the pivoted matrix columnwise. With `?latm3`, the generator is called to fill in the matrix columnwise, after which it is pivoted. Thus, `?latm3` can be used to construct random matrices which differ only in their order of rows and/or columns. `?latm2` is used to construct band matrices while avoiding calling the random number generator for entries outside the band (and therefore generating random numbers in different orders for different pivot orders).

The matrix whose (i_{sub}, j_{sub}) entry is returned is constructed as follows (this routine only computes one entry):

- If i_{sub} is outside $(1..m)$ or j_{sub} is outside $(1..n)$, returns zero (this is convenient for generating matrices in band format).
- Generate a matrix A with random entries of distribution $idist$.
- Set the diagonal to D .
- Grade the matrix, if desired, from the left (by dl) and/or from the right (by dr or dl) as specified by $igrade$.
- Permute, if desired, the rows and/or columns as specified by $ipvtng$ and $iwork$.
- Band the matrix to have lower bandwidth kl and upper bandwidth ku .
- Set random entries to zero as specified by $sparse$.

Input Parameters

<i>m</i>	Number of rows of matrix.
<i>n</i>	Number of columns of matrix.
<i>i</i>	Row of unpivoted entry to be returned.
<i>j</i>	Column of unpivoted entry to be returned.
<i>isub</i>	Row of pivoted entry to be returned.
<i>jsub</i>	Column of pivoted entry to be returned.
<i>kl</i>	Lower bandwidth.
<i>ku</i>	Upper bandwidth.
<i>idist</i>	On entry, <i>idist</i> specifies the type of distribution to be used to generate a random matrix. for <code>slatm2</code> and <code>dlatm2</code> : = 1: uniform (0,1) = 2: uniform (-1,1) = 3: normal (0,1) for <code>clatm2</code> and <code>zlatm2</code> : = 1: real and imaginary parts each uniform (0,1) = 2: real and imaginary parts each uniform (-1,1) = 3: real and imaginary parts each normal (0,1) = 4: complex number uniform in disk(0, 1)
<i>iseed</i>	Array, size 4. Seed for random number generator.
<i>d</i>	Array, size $(\min(i, j))$. Diagonal entries of matrix.

<i>igrade</i>	<p>Specifies grading of matrix as follows:</p> <ul style="list-style-type: none"> = 0: no grading = 1: matrix premultiplied by diag(<i>dl</i>) = 2: matrix postmultiplied by diag(<i>dr</i>) = 3: matrix premultiplied by diag(<i>dl</i>) and postmultiplied by diag(<i>dr</i>) = 4: matrix premultiplied by diag(<i>dl</i>) and postmultiplied by inv(diag(<i>dl</i>)) <p>For <i>slatm2</i> and <i>slatm2</i>:</p> <ul style="list-style-type: none"> = 5: matrix premultiplied by diag(<i>dl</i>) and postmultiplied by diag(<i>dl</i>) <p>For <i>clatm2</i> and <i>zlatm2</i>:</p> <ul style="list-style-type: none"> = 5: matrix premultiplied by diag(<i>dl</i>) and postmultiplied by diag(conjg(<i>dl</i>)) = 6: matrix premultiplied by diag(<i>dl</i>) and postmultiplied by diag(<i>dl</i>)
<i>dl</i>	<p>Array, size (<i>i</i> or <i>j</i>, as appropriate).</p> <p>Left scale factors for grading matrix.</p>
<i>dr</i>	<p>Array, size (<i>i</i> or <i>j</i>, as appropriate).</p> <p>Right scale factors for grading matrix.</p>
<i>ipvtng</i>	<p>On entry specifies pivoting permutations as follows:</p> <ul style="list-style-type: none"> If <i>ipvtng</i> = 0: none. If <i>ipvtng</i> = 1: row pivoting. If <i>ipvtng</i> = 2: column pivoting. If <i>ipvtng</i> = 3: full pivoting, i.e., on both sides.
<i>sparse</i>	<p>On entry, specifies the sparsity of the matrix if sparse matrix is to be generated. <i>sparse</i> should lie between 0 and 1. A uniform(0, 1) random number <i>x</i> is generated and compared to <i>sparse</i>; if <i>x</i> is larger the matrix entry is unchanged and if <i>x</i> is smaller the entry is set to zero. Thus on the average a fraction <i>sparse</i> of the entries will be set to zero.</p>
<i>iwork</i>	<p>Array, size (<i>i</i> or <i>j</i>, as appropriate). This array specifies the permutation used. The row (or column) originally in position <i>k</i> is in position <i>iwork</i>[<i>k</i> - 1] after pivoting. This differs from <i>iwork</i> for <i>?latm2</i>.</p>

Output Parameters

<i>isub</i>	On exit, row of pivoted entry is updated.
<i>jsub</i>	On exit, column of pivoted entry is updated.
<i>iseed</i>	On exit, the seed is updated.

Return Values

The function returns an entry of a random matrix (for complex variations `libmkl_gf_*` interface layer/libraries return the result as the parameter *res*).

?latm5

Generates matrices involved in the Generalized Sylvester equation.

Syntax

```
void slatm5 (*prtype, lapack_int *m, lapack_int *n, float *a, lapack_int *lda, float *b,
lapack_int *ldb, float *c, lapack_int *ldc, float *d, lapack_int *ldd, float *e,
lapack_int *lde, float *f, lapack_int *ldf, float *r, lapack_int *ldr, float *l,
lapack_int *ldl, float *alpha, lapack_int *qblcka, lapack_int *qblckb);

void dlatm5 (*prtype, lapack_int *m, lapack_int *n, double *a, lapack_int *lda, double
*b, lapack_int *ldb, double *c, lapack_int *ldc, double *d, lapack_int *ldd, double *e,
lapack_int *lde, double *f, lapack_int *ldf, double *r, lapack_int *ldr, double *l,
lapack_int *ldl, double *alpha, lapack_int *qblcka, lapack_int *qblckb);

void clatm5 (*prtype, lapack_int *m, lapack_int *n, lapack_complex_float *a, lapack_int
*lda, lapack_complex_float *b, lapack_int *ldb, lapack_complex_float *c, lapack_int
*ldc, lapack_complex_float *d, lapack_int *ldd, lapack_complex_float *e, lapack_int
*lde, lapack_complex_float *f, lapack_int *ldf, lapack_complex_float *r, lapack_int
*ldr, lapack_complex_float *l, lapack_int *ldl, float *alpha, lapack_int *qblcka,
lapack_int *qblckb);

void zlatm5 (*prtype, lapack_int *m, lapack_int *n, lapack_complex_double *a,
lapack_int *lda, lapack_complex_double *b, lapack_int *ldb, lapack_complex_double *c,
lapack_int *ldc, lapack_complex_double *d, lapack_int *ldd, lapack_complex_double *e,
lapack_int *lde, lapack_complex_double *f, lapack_int *ldf, lapack_complex_double *r,
lapack_int *ldr, lapack_complex_double *l, lapack_int *ldl, float *alpha, lapack_int
*qblcka, lapack_int *qblckb);
```

Include Files

- mkl.h

Description

The ?latm5 routine generates matrices involved in the Generalized Sylvester equation:

$$A * R - L * B = C$$

$$D * R - L * E = F$$

They also satisfy the diagonalization condition:

$$\begin{bmatrix} I & -L \\ & I \end{bmatrix} \begin{bmatrix} A & -C \\ & B \end{bmatrix} \begin{bmatrix} I & R \\ & I \end{bmatrix} = \begin{bmatrix} A & \\ & B \end{bmatrix}$$

$$\begin{bmatrix} I & -L \\ & I \end{bmatrix} \begin{bmatrix} D & -F \\ & E \end{bmatrix} \begin{bmatrix} I & R \\ & I \end{bmatrix} = \begin{bmatrix} D & \\ & E \end{bmatrix}$$

Input Parameters

prtype

Specifies the type of matrices to generate.

- If *prtype* = 1, *A* and *B* are Jordan blocks, *D* and *E* are identity matrices.

A:

If $(i == j)$ then $A_{i,j} = 1.0$.

If $(j == i + 1)$ then $A_{i,j} = -1.0$.

Otherwise $A_{i,j} = 0.0$, $i, j = 1 \dots m$

B:

If $(i == j)$ then $B_{i,j} = 1.0 - \alpha$.

If $(j == i + 1)$ then $B_{i,j} = 1.0$.

Otherwise $B_{i,j} = 0.0$, $i, j = 1 \dots n$.

D:

If $(i == j)$ then $D_{i,j} = 1.0$.

Otherwise $D_{i,j} = 0.0$, $i, j = 1 \dots m$.

E:

If $(i == j)$ then $E_{i,j} = 1.0$

Otherwise $E_{i,j} = 0.0$, $i, j = 1 \dots n$.

$L = R$ are chosen from $[-10 \dots 10]$, which specifies the right hand sides (C, F) .

- If $prtype = 2$ or 3 : Triangular and/or quasi- triangular.

A:

If $(i \leq j)$ then $A_{i,j} = [-1 \dots 1]$.

Otherwise $A_{i,j} = 0.0$, $i, j = 1 \dots M$.

If $(prtype = 3)$ then $A_{k+1,k+1} = A_{k,k}$

$A_{k+1,k} = [-1 \dots 1]$;

$\text{sign}(A_{k,k+1}) = -(\text{sign}(A_{k+1,k}))$.

$k = 1, m-1, qblocka$

B :

If $(i \leq j)$ then $B_{i,j} = [-1 \dots 1]$.

Otherwise $B_{i,j} = 0.0$, $i, j = 1 \dots n$.

If $(prtype = 3)$ then $B_{k+1,k+1} = B_{k,k}$

$B_{k+1,k} = [-1 \dots 1]$

$\text{sign}(B_{k,k+1}) = -(\text{sign}(B_{k+1,k}))$

$k = 1, n-1, qblockb$.

D:

If $(i \leq j)$ then $D_{i,j} = [-1 \dots 1]$.

Otherwise $D_{i,j} = 0.0$, $i, j = 1 \dots m$.

E:

If $(i \leq j)$ then $E_{i,j} = [-1 \dots 1]$.

Otherwise $E_{i,j} = 0.0$, $i, j = 1 \dots N$.

L, R are chosen from $[-10 \dots 10]$, which specifies the right hand sides (C, F).

- If $prtype = 4$ Full

$$A_{i,j} = [-10 \dots 10]$$

$$D_{i,j} = [-1 \dots 1] \quad i, j = 1 \dots m$$

$$B_{i,j} = [-10 \dots 10]$$

$$E_{i,j} = [-1 \dots 1] \quad i, j = 1 \dots n$$

$$R_{i,j} = [-10 \dots 10]$$

$$L_{i,j} = [-1 \dots 1] \quad i = 1 \dots m, j = 1 \dots n$$

L and R specifies the right hand sides (C, F).

- If $prtype = 5$ special case common and/or close eigs.

<i>m</i>	Specifies the order of A and D and the number of rows in C, F, R and L .
<i>n</i>	Specifies the order of B and E and the number of columns in C, F, R and L .
<i>lda</i>	The leading dimension of a .
<i>ldb</i>	The leading dimension of b .
<i>ldc</i>	The leading dimension of c .
<i>ldd</i>	The leading dimension of d .
<i>lde</i>	The leading dimension of e .
<i>ldf</i>	The leading dimension of f .
<i>ldr</i>	The leading dimension of r .
<i>ldl</i>	The leading dimension of l .
<i>alpha</i>	Parameter used in generating $prtype = 1$ and 5 matrices.
<i>qblocka</i>	When $prtype = 3$, specifies the distance between 2-by-2 blocks on the diagonal in A . Otherwise, <i>qblocka</i> is not referenced. $qblocka > 1$.
<i>qblockb</i>	When $prtype = 3$, specifies the distance between 2-by-2 blocks on the diagonal in B . Otherwise, <i>qblockb</i> is not referenced. $qblockb > 1$.

Output Parameters

<i>a</i>	Array, size $lda*m$. On exit <i>a</i> contains them-by- <i>m</i> array A initialized according to <i>prtype</i> .
<i>b</i>	Array, size $ldb*n$. On exit <i>b</i> contains the <i>n</i> -by- <i>n</i> array B initialized according to <i>prtype</i> .
<i>c</i>	Array, size $ldc*n$. On exit <i>c</i> contains the <i>m</i> -by- <i>n</i> array C initialized according to <i>prtype</i> .

<i>d</i>	Array, size <i>ldd</i> * <i>m</i> . On exit <i>d</i> contains the <i>m</i> -by- <i>m</i> array <i>D</i> initialized according to <i>prtype</i> .
<i>e</i>	Array, size <i>lde</i> * <i>n</i> . On exit <i>e</i> contains the <i>n</i> -by- <i>n</i> array <i>E</i> initialized according to <i>prtype</i> .
<i>f</i>	Array, size <i>ldf</i> * <i>n</i> . On exit <i>f</i> contains the <i>m</i> -by- <i>n</i> array <i>F</i> initialized according to <i>prtype</i> .
<i>r</i>	Array, size <i>ldr</i> * <i>n</i> . On exit <i>R</i> contains the <i>m</i> -by- <i>n</i> array <i>R</i> initialized according to <i>prtype</i> .
<i>l</i>	Array, size <i>ldl</i> * <i>n</i> . On exit <i>l</i> contains the <i>m</i> -by- <i>n</i> array <i>L</i> initialized according to <i>prtype</i> .

?latm6

Generates test matrices for the generalized eigenvalue problem, their corresponding right and left eigenvector matrices, and also reciprocal condition numbers for all eigenvalues and the reciprocal condition numbers of eigenvectors corresponding to the 1th and 5th eigenvalues.

Syntax

```
void slatm6 (lapack_int *type, lapack_int *n, float *a, lapack_int *lda, float *b, float
*x, lapack_int *ldx, float *y, lapack_int *ldy, float *alpha, float *beta, float *wx,
float *wy, float *s, float *dif);

void dlatm6 (lapack_int *type, lapack_int *n, double *a, lapack_int *lda, double *b,
double *x, lapack_int *ldx, double *y, lapack_int *ldy, double *alpha, double *beta,
double *wx, double *wy, double *s, double *dif);

void clatm6 (lapack_int *type, lapack_int *n, lapack_complex_float *a, lapack_int *lda,
lapack_complex_float *b, lapack_complex_float *x, lapack_int *ldx, lapack_complex_float
*y, lapack_int *ldy, lapack_complex_float *alpha, lapack_complex_float *beta,
lapack_complex_float *wx, lapack_complex_float *wy, float *s, float *dif);

void zlatm6 (lapack_int *type, lapack_int *n, lapack_complex_double *a, lapack_int
*lda, lapack_complex_double *b, lapack_complex_double *x, lapack_int *ldx,
lapack_complex_double *y, lapack_int *ldy, lapack_complex_double *alpha,
lapack_complex_double *beta, lapack_complex_double *wx, lapack_complex_double *wy,
double *s, double *dif);
```

Include Files

- mkl.h

Description

The ?latm6 routine generates test matrices for the generalized eigenvalue problem, their corresponding right and left eigenvector matrices, and also reciprocal condition numbers for all eigenvalues and the reciprocal condition numbers of eigenvectors corresponding to the 1th and 5th eigenvalues.

There two kinds of test matrix pairs:

$$(A, B) = \text{inverse}(YH) * (Da, Db) * \text{inverse}(X)$$

Type 1:

$$Da = \begin{bmatrix} 1+a & 0 & 0 & 0 & 0 \\ 0 & 2+a & 0 & 0 & 0 \\ 0 & 0 & 3+a & 0 & 0 \\ 0 & 0 & 0 & 4+a & 0 \\ 0 & 0 & 0 & 0 & 5+2 \end{bmatrix} \quad Db = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Type 2:

$$Da = \begin{bmatrix} 1+i & 0 & 0 & 0 & 0 \\ 0 & 1-i & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & (1+a)+(1+b)i & 0 \\ 0 & 0 & 0 & 0 & (1+a)-(1+b)i \end{bmatrix} \quad Db = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

In both cases the same `inverse(YH)` and `inverse(X)` are used to compute (A, B) , giving the exact eigenvectors to (A, B) as (YH, X) :

$$YH = \begin{bmatrix} 1 & 0 & -y & y & -y \\ 0 & 1 & -y & y & -y \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad X = \begin{bmatrix} 1 & 0 & -x & -x & x \\ 0 & 1 & x & -x & -x \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

,

where a, b, x and y will have all values independently of each other.

Input Parameters

<code>type</code>	Specifies the problem type.
<code>n</code>	Size of the matrices A and B .
<code>lda</code>	The leading dimension of a and of b .
<code>ldx</code>	The leading dimension of x .
<code>ldy</code>	The leading dimension of y .
<code>alpha, beta</code>	Weighting constants for matrix A .
<code>wx</code>	Constant for right eigenvector matrix.
<code>wy</code>	Constant for left eigenvector matrix.

Output Parameters

<code>a</code>	Array, size <code>lda*n</code> . On exit, a contains the n -by- n matrix initialized according to <code>type</code> .
----------------	---

<i>b</i>	Array, size <i>lda</i> * <i>n</i> . On exit, <i>b</i> contains the <i>n</i> -by- <i>n</i> matrix initialized according to <i>type</i> .
<i>x</i>	Array, size <i>ldx</i> * <i>n</i> . On exit, <i>x</i> contains the <i>n</i> -by- <i>n</i> matrix of right eigenvectors.
<i>y</i>	Array, size <i>ldy</i> * <i>n</i> . On exit, <i>y</i> is the <i>n</i> -by- <i>n</i> matrix of left eigenvectors.
<i>s</i>	Array, size (<i>n</i>). <i>s</i> [<i>i</i> - 1] is the reciprocal condition number for eigenvalue <i>i</i> .
<i>dif</i>	Array, size(<i>n</i>). <i>dif</i> [<i>i</i> - 1] is the reciprocal condition number for eigenvector <i>i</i> .

?latme

Generates random non-symmetric square matrices with specified eigenvalues.

Syntax

```
void slatme (lapack_int *n, char *dist, lapack_int *iseed, float *d, lapack_int *mode,
float *cond, float *dmax, char *ei, char *rsign, char *upper, char *sim, float *ds,
lapack_int *modes, float *conds, lapack_int *kl, lapack_int *ku, float *anorm, float *a,
lapack_int *lda, float *work, lapack_int *info); void dlatme (lapack_int *n, char *dist,
lapack_int *iseed, double *d, lapack_int *mode, double *cond, double *dmax, char *ei,
char *rsign, char *upper, char *sim, double *ds, lapack_int *modes, double *conds,
lapack_int *kl, lapack_int *ku, double *anorm, double *a, lapack_int *lda, double *work,
lapack_int *info); void clatme (lapack_int *n, char *dist, lapack_int *iseed,
lapack_complex_float *d, lapack_int *mode, float *cond, lapack_complex_float *dmax,
char *ei, char *rsign, char *upper, char *sim, float *ds, lapack_int *modes, float
*conds, lapack_int *kl, lapack_int *ku, float *anorm, lapack_complex_float *a,
lapack_int *lda, lapack_complex_float *work, lapack_int *info); void zlatme (lapack_int
*n, char *dist, lapack_int *iseed, lapack_complex_double *d, lapack_int *mode, double
*cond, lapack_complex_double *dmax, char *ei, char *rsign, char *upper, char *sim,
double *ds, lapack_int *modes, double *conds, lapack_int *kl, lapack_int *ku, double
*anorm, lapack_complex_double *a, lapack_int *lda, lapack_complex_double *work,
lapack_int *info);
```

Include Files

- mkl.h

Description

The ?latme routine generates random non-symmetric square matrices with specified eigenvalues. ?latme operates by applying the following sequence of operations:

1. Set the diagonal to *d*, where *d* may be input or computed according to *mode*, *cond*, *dmax*, and *rsign* as described below.
2. If *upper* = 'T', the upper triangle of *a* is set to random values out of distribution *dist*.
3. If *sim*='T', *a* is multiplied on the left by a random matrix *X*, whose singular values are specified by *ds*, *modes*, and *conds*, and on the right by *X* inverse.
4. If *kl* < *n*-1, the lower bandwidth is reduced to *kl* using Householder transformations. If *ku* < *n*-1, the upper bandwidth is reduced to *ku*.

5. If *anorm* is not negative, the matrix is scaled to have maximum-element-norm *anorm*.

NOTE

Since the matrix cannot be reduced beyond Hessenberg form, no packing options are available.

Input Parameters

<i>n</i>	The number of columns (or rows) of <i>A</i> .
<i>dist</i>	<p>On entry, <i>dist</i> specifies the type of distribution to be used to generate the random eigen-/singular values, and on the upper triangle (see <i>upper</i>).</p> <p>If <i>dist</i> = 'U': uniform(0, 1)</p> <p>If <i>dist</i> = 'S': uniform(-1, 1)</p> <p>If <i>dist</i> = 'N': normal(0, 1)</p> <p>If <i>dist</i> = 'D': uniform on the complex disc $z < 1$.</p>
<i>iseed</i>	<p>Array, size 4.</p> <p>On entry <i>iseed</i> specifies the seed of the random number generator. The elements should lie between 0 and 4095 inclusive, and <i>iseed</i>[3] should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers.</p>
<i>d</i>	<p>Array, size (<i>n</i>). This array is used to specify the eigenvalues of <i>A</i>.</p> <p>If <i>mode</i> = 0, then <i>d</i> is assumed to contain the eigenvalues. Otherwise they are computed according to <i>mode</i>, <i>cond</i>, <i>dmax</i>, and <i>rsign</i> and placed in <i>d</i>.</p>
<i>mode</i>	<p>On entry <i>mode</i> describes how the eigenvalues are to be specified:</p> <p><i>mode</i> = 0 means use <i>d</i> (with <i>ei</i> for <i>slatme</i> and <i>dlatme</i>) as input.</p> <p><i>mode</i> = 1 sets $d[0] = 1$ and $d(2:n)=1.0/cond$.</p> <p><i>mode</i> = 2 sets $d[0:n - 2] = 1$ and $d[n - 1]=1.0/cond$.</p> <p><i>mode</i> = 3 sets $d[i - 1] = cond**(-(i-1)/(n-1))$.</p> <p><i>mode</i> = 4 sets $d[i - 1] = 1 - (i-1)/(n-1)*(1 - 1/cond)$.</p> <p><i>mode</i> = 5 sets <i>d</i> to random numbers in the range ($1/cond$, 1) such that their logarithms are uniformly distributed.</p> <p><i>mode</i> = 6 sets <i>d</i> to random numbers from same distribution as the rest of the matrix.</p> <p><i>mode</i> < 0 has the same meaning as $abs(mode)$, except that the order of the elements of <i>d</i> is reversed.</p> <p>Thus if <i>mode</i> is between 1 and 4, <i>d</i> has entries ranging from 1 to $1/cond$, if between -1 and -4, <i>d</i> has entries ranging from $1/cond$ to 1.</p>
<i>cond</i>	On entry, this is used as described under <i>mode</i> above. If used, it must be ≥ 1 .

<i>dmax</i>	<p>If <i>mode</i> is not -6, 0 or 6, the contents of <i>d</i> as computed according to <i>mode</i> and <i>cond</i> are scaled by $dmax / \max(\text{abs}(d[i - 1]))$. Note that <i>dmax</i> needs not be positive or real: if <i>dmax</i> is negative or complex (or zero), <i>d</i> will be scaled by a negative or complex number (or zero). If <i>rsign</i>='F' then the largest (absolute) eigenvalue will be equal to <i>dmax</i>.</p>
<i>ei</i>	<p>Used by <i>slatme</i> and <i>dlatme</i> only.</p> <p>Array, size (<i>n</i>).</p> <p>If <i>mode</i> = 0, and <i>ei</i>[0] is not ' ' (space character), this array specifies which elements of <i>d</i> (on input) are real eigenvalues and which are the real and imaginary parts of a complex conjugate pair of eigenvalues. The elements of <i>ei</i> may then only have the values 'R' and 'I'.</p> <p>If <i>ei</i>[<i>j</i> - 1] = 'R' and <i>ei</i>[<i>j</i>] = 'I', then the <i>j</i>-th eigenvalue is $\text{cmplx}(d[j - 1], d[j])$, and the (<i>j</i> + 1)-th is the complex conjugate thereof.</p> <p>If <i>ei</i>[<i>j</i> - 1] = <i>ei</i>[<i>j</i>] = 'R', then the <i>j</i>-th eigenvalue is <i>d</i>[<i>j</i> - 1] (i.e., real). <i>ei</i>[0] may not be 'I', nor may two adjacent elements of <i>ei</i> both have the value 'I'.</p> <p>If <i>mode</i> is not 0, then <i>ei</i> is ignored. If <i>mode</i> is 0 and <i>ei</i>[0] = ' ', then the eigenvalues will all be real.</p>
<i>rsign</i>	<p>If <i>mode</i> is not 0, 6, or -6, and <i>rsign</i> = 'T', then the elements of <i>d</i>, as computed according to <i>mode</i> and <i>cond</i>, are multiplied by a random sign (+1 or -1) for <i>slatme</i> and <i>dlatme</i> or by a complex number from the unit circle $z = 1$ for <i>clatme</i> and <i>zlatme</i>.</p> <p>If <i>rsign</i> = 'F', the elements of <i>d</i> are not multiplied. <i>rsign</i> may only have the values 'T' or 'F'.</p>
<i>upper</i>	<p>If <i>upper</i> = 'T', then the elements of <i>a</i> above the diagonal will be set to random numbers out of <i>dist</i>.</p> <p>If <i>upper</i> = 'F', they will not. <i>upper</i> may only have the values 'T' or 'F'.</p>
<i>sim</i>	<p>If <i>sim</i> = 'T', then <i>a</i> will be operated on by a "similarity transform", i.e., multiplied on the left by a matrix <i>X</i> and on the right by <i>X</i> inverse. $X = USV$, where <i>U</i> and <i>V</i> are random unitary matrices and <i>S</i> is a (diagonal) matrix of singular values specified by <i>ds</i>, <i>modes</i>, and <i>conds</i>.</p> <p>If <i>sim</i> = 'F', then <i>a</i> will not be transformed.</p>
<i>ds</i>	<p>This array is used to specify the singular values of <i>X</i>, in the same way that <i>d</i> specifies the eigenvalues of <i>a</i>. If <i>mode</i> = 0, the <i>ds</i> contains the singular values, which may not be zero.</p>
<i>modes</i>	<p>Similar to <i>mode</i>, but for specifying the diagonal of <i>S</i>. <i>modes</i> = -6 and +6 are not allowed (since they would result in randomly ill-conditioned eigenvalues.)</p>
<i>conds</i>	<p>Similar to <i>cond</i>, but for specifying the diagonal of <i>S</i>.</p>

<i>kl</i>	This specifies the lower bandwidth of the matrix. <i>kl</i> = 1 specifies upper Hessenberg form. If <i>kl</i> is at least $n-1$, then <i>A</i> will have full lower bandwidth.
<i>ku</i>	This specifies the upper bandwidth of the matrix. <i>ku</i> = 1 specifies lower Hessenberg form. If <i>ku</i> is at least $n-1$, then <i>a</i> will have full upper bandwidth. If <i>ku</i> and <i>kl</i> are both at least $n-1$, then <i>a</i> will be dense. Only one of <i>ku</i> and <i>kl</i> may be less than $n-1$.
<i>anorm</i>	If <i>anorm</i> is not negative, then <i>a</i> is scaled by a non-negative real number to make the maximum-element-norm of <i>a</i> to be <i>anorm</i> .
<i>lda</i>	Number of rows of matrix <i>A</i> .
<i>work</i>	Array, size $(3*n)$. Workspace.

Output Parameters

<i>iseed</i>	On exit, the seed is updated.
<i>d</i>	Modified if <i>mode</i> is nonzero.
<i>ds</i>	Modified if <i>mode</i> is nonzero.
<i>a</i>	Array, size $lda*n$. On exit, <i>a</i> is the desired test matrix.
<i>info</i>	If <i>info</i> = 0, execution is successful. If <i>info</i> = -1, <i>n</i> is negative . If <i>info</i> = -2, <i>dist</i> is an illegal string. If <i>info</i> = -5, <i>mode</i> is not in range -6 to 6. If <i>info</i> = -6, <i>cond</i> is less than 1.0, and <i>mode</i> is not -6, 0, or 6 . If <i>info</i> = -9, <i>rsign</i> is not 'T' or 'F' . If <i>info</i> = -10, <i>upper</i> is not 'T' or 'F'. If <i>info</i> = -11, <i>sim</i> is not 'T' or 'F'. If <i>info</i> = -12, <i>modes</i> = 0 and <i>ds</i> has a zero singular value. If <i>info</i> = -13, <i>modes</i> is not in the range -5 to 5. If <i>info</i> = -14, <i>modes</i> is nonzero and <i>conds</i> is less than 1. . If <i>info</i> = -15, <i>kl</i> is less than 1. If <i>info</i> = -16, <i>ku</i> is less than 1, or <i>kl</i> and <i>ku</i> are both less than $n-1$. If <i>info</i> = -19, <i>lda</i> is less than <i>m</i> . If <i>info</i> = 1, error return from ?latm1 (computing <i>d</i>) . If <i>info</i> = 2, cannot scale to <i>dmax</i> (max. eigenvalue is 0) . If <i>info</i> = 3, error return from slatm1(for slatme and clatme), dlatm1 (for dlatme and zlatme) . If <i>info</i> = 4, error return from ?large.

If *info* = 5, zero singular value from slatml(for slatme and clatme), dlatml(for dlatme and zlatme).

?latmr

Generates random matrices of various types.

Syntax

```
void slatmr (lapack_int *m, lapack_int *n, char *dist, lapack_int *iseed, char *sym,
float *d, lapack_int *mode, float *cond, float *dmax, char *rsign, char *grade, float
*d1, lapack_int *model, float *cond1, float *dr, lapack_int *moder, float *condr, char
*pivtnng, lapack_int *ipivot, lapack_int *kl, lapack_int *ku, float *sparse, float
*anorm, char *pack, float *a, lapack_int *lda, lapack_int *iwork, lapack_int *info);

void dlatmr (lapack_int *m, lapack_int *n, char *dist, lapack_int *iseed, char *sym,
double *d, lapack_int *mode, double *cond, double *dmax, char *rsign, char *grade,
double *d1, lapack_int *model, double *cond1, double *dr, lapack_int *moder, double
*condr, char *pivtnng, lapack_int *ipivot, lapack_int *kl, lapack_int *ku, double
*sparse, double *anorm, char *pack, double *a, lapack_int *lda, lapack_int *iwork,
lapack_int *info);

void clatmr (lapack_int *m, lapack_int *n, char *dist, lapack_int *iseed, char *sym,
lapack_complex *d, lapack_int *mode, float *cond, lapack_complex *dmax, char *rsign,
char *grade, lapack_complex *d1, lapack_int *model, float *cond1, lapack_complex *dr,
lapack_int *moder, float *condr, char *pivtnng, lapack_int *ipivot, lapack_int *kl,
lapack_int *ku, float *sparse, float *anorm, char *pack, float *a, lapack_int *lda,
lapack_int *iwork, lapack_int *info);

void zlatmr (lapack_int *m, lapack_int *n, char *dist, lapack_int *iseed, char *sym,
lapack_complex_double *d, lapack_int *mode, float *cond, lapack_complex_double *dmax,
char *rsign, char *grade, lapack_complex_double *d1, lapack_int *model, float *cond1,
lapack_complex_double *dr, lapack_int *moder, float *condr, char *pivtnng, lapack_int
*ipivot, lapack_int *kl, lapack_int *ku, float *sparse, float *anorm, char *pack, float
*a, lapack_int *lda, lapack_int *iwork, lapack_int *info);
```

Description

The ?latmr routine operates by applying the following sequence of operations:

1. Generate a matrix *A* with random entries of distribution *dist*:
 If *sym* = 'S', the matrix is symmetric,
 If *sym* = 'H', the matrix is Hermitian,
 If *sym* = 'N', the matrix is nonsymmetric.
2. Set the diagonal to *D*, where *D* may be input or computed according to *mode*, *cond*, *dmax* and *rsign* as described below.
3. Grade the matrix, if desired, from the left or right as specified by *grade*. The inputs *d1*, *model*, *cond1*, *dr*, *moder* and *condr* also determine the grading as described below.
4. Permute, if desired, the rows and/or columns as specified by *pivtnng* and *ipivot*.
5. Set random entries to zero, if desired, to get a random sparse matrix as specified by *sparse*.
6. Make *A* a band matrix, if desired, by zeroing out the matrix outside a band of lower bandwidth *kl* and upper bandwidth *ku*.
7. Scale *A*, if desired, to have maximum entry *anorm*.

8. Pack the matrix if desired. See options specified by the *pack* parameter.

NOTE

If two calls to `?latmr` differ only in the *pack* parameter, they generate mathematically equivalent matrices. If two calls to `?latmr` both have full bandwidth ($kl = m-1$ and $ku = n-1$), and differ only in the *pivtn*g and *pack* parameters, then the matrices generated differ only in the order of the rows and columns, and otherwise contain the same data. This consistency cannot be and is not maintained with less than full bandwidth.

Input Parameters

<i>m</i>	Number of rows of A.
<i>n</i>	Number of columns of A.
<i>dist</i>	On entry, <i>dist</i> specifies the type of distribution to be used to generate a random matrix . If <i>dist</i> = 'U', real and imaginary parts are independent uniform(0, 1). If <i>dist</i> = 'S', real and imaginary parts are independent uniform(-1, 1). If <i>dist</i> = 'N', real and imaginary parts are independent normal(0, 1). If <i>dist</i> = 'D', distribution is uniform on interior of unit disk.
<i>iseed</i>	Array, size 4. On entry, <i>iseed</i> specifies the seed of the random number generator. They should lie between 0 and 4095 inclusive, and <i>iseed</i> [3] should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers.
<i>sym</i>	If <i>sym</i> = 'S', generated matrix is symmetric. If <i>sym</i> = 'H', generated matrix is Hermitian. If <i>sym</i> = 'N', generated matrix is nonsymmetric.
<i>d</i>	On entry this array specifies the diagonal entries of the diagonal of A. <i>d</i> may either be specified on entry, or set according to <i>mode</i> and <i>cond</i> as described below. If the matrix is Hermitian, the real part of <i>d</i> is taken. May be changed on exit if <i>mode</i> is nonzero.
<i>mode</i>	On entry describes how <i>d</i> is to be used: <i>mode</i> = 0 means use <i>d</i> as input. <i>mode</i> = 1 sets <i>d</i> [0]=1 and <i>d</i> [1: <i>n</i> - 1]=1.0/ <i>cond</i> . <i>mode</i> = 2 sets <i>d</i> [0: <i>n</i> - 2]=1 and <i>d</i> [<i>n</i> - 1]=1.0/ <i>cond</i> . <i>mode</i> = 3 sets <i>d</i> [<i>i</i> - 1]= <i>cond</i> **(-(i-1)/(n-1)). <i>mode</i> = 4 sets <i>d</i> [<i>i</i> - 1]=1 - (i-1)/(n-1)*(1 - 1/ <i>cond</i>). <i>mode</i> = 5 sets <i>d</i> to random numbers in the range (1/ <i>cond</i> , 1) such that their logarithms are uniformly distributed.

$mode = 6$ sets d to random numbers from same distribution as the rest of the matrix.

$mode < 0$ has the same meaning as $abs(mode)$, except that the order of the elements of d is reversed.

Thus if $mode$ is between 1 and 4, d has entries ranging from 1 to $1/cond$, if between -1 and -4, D has entries ranging from $1/cond$ to 1.

cond

On entry, used as described under *mode* above. If used, *cond* must be ≥ 1 .

dmax

If *mode* is not -6, 0, or 6, the diagonal is scaled by $dmax / \max(abs(d[i]))$, so that maximum absolute entry of diagonal is $abs(dmax)$. If *dmax* is complex (or zero), the diagonal is scaled by a complex number (or zero).

rsign

If *mode* is not -6, 0, or 6, specifies the sign of the diagonal as follows:

For *slatmr* and *dlatmr*, if *rsign* = 'T', diagonal entries are multiplied 1 or -1 with a probability of 0.5.

For *clatmr* and *zlatmr*, if *rsign* = 'T', diagonal entries are multiplied by a random complex number uniformly distributed with absolute value 1.

If *rsign* = 'F', diagonal entries are unchanged.

grade

Specifies grading of matrix as follows:

If *grade* = 'N', there is no grading

If *grade* = 'L', matrix is premultiplied by $diag(dl)$ (only if matrix is nonsymmetric)

If *grade* = 'R', matrix is postmultiplied by $diag(dr)$ (only if matrix is nonsymmetric)

If *grade* = 'B', matrix is premultiplied by $diag(dl)$ and postmultiplied by $diag(dr)$ (only if matrix is nonsymmetric)

If *grade* = 'H', matrix is premultiplied by $diag(dl)$ and postmultiplied by $diag(conjg(dl))$ (only if matrix is Hermitian or nonsymmetric)

If *grade* = 'S', matrix is premultiplied by $diag(dl)$ and postmultiplied by $diag(dl)$ (only if matrix is symmetric or nonsymmetric)

If *grade* = 'E', matrix is premultiplied by $diag(dl)$ and postmultiplied by $inv(diag(dl))$ (only if matrix is nonsymmetric)

NOTE

if *grade* = 'E', then m must equal n .

dl

Array, size (m).

If *model* = 0, then on entry this array specifies the diagonal entries of a diagonal matrix used as described under *grade* above.

If *model* is not zero, then *dl* is set according to *model* and *condl*, analogous to the way D is set according to *mode* and *cond* (except there is no *dmax* parameter for *dl*).

If *grade* = 'E', then *dl* cannot have zero entries.

	Not referenced if <i>grade</i> = 'N' or 'R'. Changed on exit.
<i>model</i>	This specifies how the diagonal array <i>dl</i> is computed, just as <i>mode</i> specifies how <i>D</i> is computed.
<i>condl</i>	When <i>model</i> is not zero, this specifies the condition number of the computed <i>dl</i> .
<i>dr</i>	<p>If <i>moder</i> = 0, then on entry this array specifies the diagonal entries of a diagonal matrix used as described under <i>grade</i> above.</p> <p>If <i>moder</i> is not zero, then <i>dr</i> is set according to <i>moder</i> and <i>condr</i>, analogous to the way <i>d</i> is set according to <i>mode</i> and <i>cond</i> (except there is no <i>dmax</i> parameter for <i>dr</i>).</p> <p>Not referenced if <i>grade</i> = 'N', 'L', 'H', 'S' or 'E'.</p>
<i>moder</i>	This specifies how the diagonal array <i>dr</i> is to be computed, just as <i>mode</i> specifies how <i>d</i> is to be computed.
<i>condr</i>	When <i>moder</i> is not zero, this specifies the condition number of the computed <i>dr</i> .
<i>pivtnng</i>	<p>On entry specifies pivoting permutations as follows:</p> <p>If <i>pivtnng</i> = 'N' or ' ': no pivoting permutation.</p> <p>If <i>pivtnng</i> = 'L': left or row pivoting (matrix must be nonsymmetric).</p> <p>If <i>pivtnng</i> = 'R': right or column pivoting (matrix must be nonsymmetric).</p> <p>If <i>pivtnng</i> = 'B' or 'F': both or full pivoting, i.e., on both sides. In this case, <i>m</i> must equal <i>n</i>.</p> <p>If two calls to ?latmr both have full bandwidth ($kl = m - 1$ and $ku = n - 1$), and differ only in the <i>pivtnng</i> and <i>pack</i> parameters, then the matrices generated differs only in the order of the rows and columns, and otherwise contain the same data. This consistency cannot be maintained with less than full bandwidth.</p>
<i>ipivot</i>	<p>Array, size (<i>n</i> or <i>m</i>) This array specifies the permutation used. After the basic matrix is generated, the rows, columns, or both are permuted.</p> <p>If row pivoting is selected, ?latmr starts with the last row and interchanges row <i>m</i> and row <i>ipivot</i>[<i>m</i> - 1], then moves to the next-to-last row, interchanging rows [<i>m</i> - 2] and row <i>ipivot</i>[<i>m</i> - 2], and so on. In terms of "2-cycles", the permutation is (1 <i>ipivot</i>[0]) (2 <i>ipivot</i>[1]) ... (<i>mipivot</i>[<i>m</i> - 1]) where the rightmost cycle is applied first. This is the inverse of the effect of pivoting in LINPACK. The idea is that factoring (with pivoting) an identity matrix which has been inverse-pivoted in this way should result in a pivot vector identical to <i>ipivot</i>. Not referenced if <i>pivtnng</i> = 'N'.</p>
<i>sparse</i>	On entry, specifies the sparsity of the matrix if a sparse matrix is to be generated. <i>sparse</i> should lie between 0 and 1. To generate a sparse matrix, for each matrix entry a uniform (0, 1) random number <i>x</i> is generated and compared to <i>sparse</i> ; if <i>x</i> is larger the matrix entry is unchanged and if <i>x</i> is smaller the entry is set to zero. Thus on the average a fraction <i>sparse</i> of the entries is set to zero.

- kl* On entry, specifies the lower bandwidth of the matrix. For example, $kl = 0$ implies upper triangular, $kl = 1$ implies upper Hessenberg, and kl at least $m-1$ implies the matrix is not banded. Must equal ku if matrix is symmetric or Hermitian.
- ku* On entry, specifies the upper bandwidth of the matrix. For example, $ku = 0$ implies lower triangular, $ku = 1$ implies lower Hessenberg, and ku at least $n-1$ implies the matrix is not banded. Must equal kl if matrix is symmetric or Hermitian.
- anorm* On entry, specifies maximum entry of output matrix (output matrix is multiplied by a constant so that its largest absolute entry equal *anorm*) if *anorm* is nonnegative. If *anorm* is negative no scaling is done.
- pack* On entry, specifies packing of matrix as follows:
- If *pack* = 'N': no packing
 - If *pack* = 'U': zero out all subdiagonal entries (if symmetric or Hermitian)
 - If *pack* = 'L': zero out all superdiagonal entries (if symmetric or Hermitian)
 - If *pack* = 'C': store the upper triangle columnwise (only if matrix symmetric or Hermitian or square upper triangular)
 - If *pack* = 'R': store the lower triangle columnwise (only if matrix symmetric or Hermitian or square lower triangular) (same as upper half rowwise if symmetric) (same as conjugate upper half rowwise if Hermitian)
 - If *pack* = 'B': store the lower triangle in band storage scheme (only if matrix symmetric or Hermitian)
 - If *pack* = 'Q': store the upper triangle in band storage scheme (only if matrix symmetric or Hermitian)
 - If *pack* = 'Z': store the entire matrix in band storage scheme (pivoting can be provided for by using this option to store *A* in the trailing rows of the allocated storage)
- Using these options, the various LAPACK packed and banded storage schemes can be obtained:

LAPACK storage scheme	Value of <i>pack</i>
GB	'Z'
PB, HB or TB	'B' or 'Q'
PP, HP or TP	'C' or 'R'

If two calls to ?latmr differ only in the pack parameter, they generate mathematically equivalent matrices.

- lda* On entry, *lda* specifies the first dimension of *a* as declared in the calling program.
- If *pack* = 'N', 'U' or 'L', *lda* must be at least $\max(1, m)$.
 - If *pack* = 'C' or 'R', *lda* must be at least 1.
 - If *pack* = 'B', or 'Q', *lda* must be $\min(ku + 1, n)$.

If *pack* = 'Z', *lda* must be at least $kuu + kll + 1$, where $kuu = \min(ku, n-1)$ and $kll = \min(kl, n-1)$.

iwork

Array, size (*n* or *m*). Workspace. Not referenced if *pivtnng* = 'N'. Changed on exit.

Output Parameters

iseed

On exit, the seed is changed.

d

May be changed on exit if *mode* is nonzero.

dl

On exit, array is changed.

dr

On exit, array is changed.

a

On exit, *a* is the desired test matrix. Only those entries of *a* which are significant on output is referenced (even if *a* is in packed or band storage format). The unoccupied corners of *a* in band format are zeroed out.

info

If *info* = 0, the execution is successful.

If *info* = -1, *m* is negative or unequal to *n* and *sym* = 'S' or 'H'.

If *info* = -2, *n* is negative .

If *info* = -3, *dist* is an illegal string.

If *info* = -5, *sym* is an illegal string..

If *info* = -7, *mode* is not in range -6 to 6.

If *info* = -8, *cond* is less than 1.0, and *mode* is neither -6, 0 nor 6.

If *info* = -10, *mode* is neither -6, 0 nor 6 and *rsign* is an illegal string.

If *info* = -11, *grade* is an illegal string, or *grade* = 'E' and *m* is not equal to *n*, or *grade*='L', 'R', 'B', 'S' or 'E' and *sym* = 'H', or *grade* = 'L', 'R', 'B', 'H' or 'E' and *sym* = 'S'

If *info* = -12, *grade* = 'E' and *dl* contains zero .

If *info* = -13, *model* is not in range -6 to 6 and *grade* = 'L', 'B', 'H', 'S' or 'E' .

If *info* = -14, *concl* is less than 1.0, *grade* = 'L', 'B', 'H', 'S' or 'E', and *model* is neither -6, 0 nor 6.

If *info* = -16, *moder* is not in range -6 to 6 and *grade* = 'R' or 'B' .

If *info* = -17, *condr* is less than 1.0, *grade* = 'R' or 'B', and *moder* is neither -6, 0 nor 6 .

If *info* = -18, *pivtnng* is an illegal string, or *pivtnng* = 'B' or 'F' and *m* is not equal to *n*, or *pivtnng* = 'L' or 'R' and *sym* = 'S' or 'H'.

If *info* = -19, *ipivot* contains out of range number and *pivtnng* is not equal to 'N' .

If *info* = -20, *kl* is negative.

If *info* = -21, *ku* is negative, or *sym* = 'S' or 'H' and *ku* not equal to *kl*.

If *info* = -22, *sparse* is not in range 0 to 1.

If *info* = -24, *pack* is an illegal string, or *pack* = 'U', 'L', 'B' or 'Q' and *sym* = 'N', or *pack* = 'C' and *sym* = 'N' and either *kl* is not equal to 0 or *n* is not equal to *m*, or *pack* = 'R' and *sym* = 'N', and either *ku* is not equal to 0 or *n* is not equal to *m*.

If *info* = -26, *lda* is too small.

If *info* = 1, error return from ?latml (computing *D*).

If *info* = 2, cannot scale to *dmax* (max. entry is 0).

If *info* = 3, error return from ?latml (computing *dl*).

If *info* = 4, error return from ?latml (computing *dr*).

If *info* = 5, *anorm* is positive, but matrix constructed prior to attempting to scale it to have norm *anorm*, is zero.

?lauum

Computes the product $U*U^T(U*U^H)$ or $L^T*L(L^H*L)$, where *U* and *L* are upper or lower triangular matrices (blocked algorithm).

Syntax

```
lapack_int LAPACKE_slauum (int matrix_layout , char uplo , lapack_int n , float * a ,
lapack_int lda );

lapack_int LAPACKE_dlauum (int matrix_layout , char uplo , lapack_int n , double * a ,
lapack_int lda );

lapack_int LAPACKE_clauum (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda );

lapack_int LAPACKE_zlauum (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda );
```

Include Files

- mkl.h

Description

The routine ?lauum computes the product $U*U^T$ or L^T*L for real flavors, and $U*U^H$ or L^H*L for complex flavors. Here the triangular factor *U* or *L* is stored in the upper or lower triangular part of the array *a*.

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor *U* in *A*.

If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor *L* in *A*.

This is the blocked form of the algorithm, calling [BLAS Level 3 Routines](#).

Input Parameters

A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>uplo</i>	Specifies whether the triangular factor stored in the array <i>a</i> is upper or lower triangular: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	The order of the triangular factor <i>U</i> or <i>L</i> . $n \geq 0$.
<i>a</i>	Array of size $\max(1, lda * n)$. On entry, the triangular factor <i>U</i> or <i>L</i> .
<i>lda</i>	The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.

Output Parameters

<i>a</i>	On exit, if <i>uplo</i> = 'U', then the upper triangle of <i>a</i> is overwritten with the upper triangle of the product $U * U^T (U * U^H)$; if <i>uplo</i> = 'L', then the lower triangle of <i>a</i> is overwritten with the lower triangle of the product $L^T * L (L^H * L)$.
----------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*k*, the *k*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?syswapr

Applies an elementary permutation on the rows and columns of a symmetric matrix.

Syntax

```
lapack_int LAPACKE_ssyswapr (int matrix_layout , char uplo , lapack_int n , float * a ,
lapack_int i1 , lapack_int i2 );

lapack_int LAPACKE_dsyswapr (int matrix_layout , char uplo , lapack_int n , double *
a , lapack_int i1 , lapack_int i2 );

lapack_int LAPACKE_csyswapr (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int i1 , lapack_int i2 );

lapack_int LAPACKE_zsyswapr (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int i1 , lapack_int i2 );
```

Include Files

- mkl.h

Description

The routine applies an elementary permutation on the rows and columns of a symmetric matrix.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix <i>A</i> has been factored: If <code>uplo = 'U'</code> , the array <i>a</i> stores the upper triangular factor <i>U</i> of the factorization $A = U * D * U^T$. If <code>uplo = 'L'</code> , the array <i>a</i> stores the lower triangular factor <i>L</i> of the factorization $A = L * D * L^T$.
<code>n</code>	The order of matrix <i>A</i> ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides; $nrhs \geq 0$.
<code>a</code>	Array of size at least $\max(1, lda * n)$. The array <i>a</i> contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> as computed by <code>?sytrf</code> .
<code>i1</code>	Index of the first row to swap.
<code>i2</code>	Index of the second row to swap.

Output Parameters

<code>a</code>	If <code>info = 0</code> , the symmetric inverse of the original matrix. If <code>info = 'U'</code> , the upper triangular part of the inverse is formed and the part of <i>A</i> below the diagonal is not referenced. If <code>info = 'L'</code> , the lower triangular part of the inverse is formed and the part of <i>A</i> above the diagonal is not referenced.
----------------	--

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, the *i*-th parameter had an illegal value.

If `info = -1011`, memory allocation error occurred.

See Also

[?sytrf](#)

?heswapr

Applies an elementary permutation on the rows and columns of a Hermitian matrix.

Syntax

```
lapack_int LAPACKE_cheswapr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float* a, lapack_int i1, lapack_int i2);
```

```
lapack_int LAPACKE_zheswapr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double* a, lapack_int i1, lapack_int i2);
```

Include Files

- mkl.h

Description

The routine applies an elementary permutation on the rows and columns of a Hermitian matrix.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <code>uplo</code> = 'U', the array a stores the upper triangular factor U of the factorization $A = U * D * U^H$. If <code>uplo</code> = 'L', the array a stores the lower triangular factor L of the factorization $A = L * D * L^H$.
<code>n</code>	The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	The number of right-hand sides; $nrhs \geq 0$.
<code>a</code>	Array of size at least <code>max(1, lda*n)</code> . The array a contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <code>?hetrf</code> .
<code>i1</code>	Index of the first row to swap.
<code>i2</code>	Index of the second row to swap.

Output Parameters

<code>a</code>	If <code>info</code> = 0, the inverse of the original matrix. If <code>info</code> = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If <code>info</code> = 'L', the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
----------------	---

Return Values

This function returns a value `info`.

If `info` = 0, the execution is successful.

If `info` = $-i$, the i -th parameter had an illegal value.

If `info` = -1011, memory allocation error occurred.

See Also

[?hetrf](#)

?sfrk

Performs a symmetric rank- k operation for matrix in RFP format.

Syntax

```
lapack_int LAPACKESsfrk (int matrix_layout , char transr , char uplo , char trans ,
lapack_int n , lapack_int k , float alpha , const float * a , lapack_int lda , float
beta , float * c );
```

```
lapack_int LAPACKEdsfrk (int matrix_layout , char transr , char uplo , char trans ,
lapack_int n , lapack_int k , double alpha , const double * a , lapack_int lda , double
beta , double * c );
```

Include Files

- mkl.h

Description

The `?sfrk` routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

$$C := \alpha * A * A^T + \beta * C,$$

or

$$C := \alpha * A^T * A + \beta * C,$$

where:

α and β are scalars,

C is an n -by- n symmetric matrix in [rectangular full packed \(RFP\) format](#),

A is an n -by- k matrix in the first case and a k -by- n matrix in the second case.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>transr</i>	if <i>transr</i> = 'N' or 'n', the normal form of RFP C is stored; if <i>transr</i> = 'T' or 't', the transpose form of RFP C is stored.
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array c is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array c is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array c is used.
<i>trans</i>	Specifies the operation: if <i>trans</i> = 'N' or 'n', then $C := \alpha * A * A^T + \beta * C$; if <i>trans</i> = 'T' or 't', then $C := \alpha * A^T * A + \beta * C$;
<i>n</i>	Specifies the order of the matrix C . The value of n must be at least zero.

k On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the matrix *A*, and on entry with *trans* = 'T' or 't', *k* specifies the number of rows of the matrix *A*.

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

alpha Specifies the scalar *alpha*.

a Array, size $\max(1, lda * ka)$, where *ka* is in the following table:

	Col_major	Row_major
<i>trans</i> = 'N'	<i>k</i>	<i>n</i>
<i>trans</i> = 'T'	<i>n</i>	<i>k</i>

Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda Specifies the leading dimension of *a* as declared in the calling (sub)program. *lda* is defined by the following table:

	Col_major	Row_major
<i>trans</i> = 'N'	$\max(1, n)$	$\max(1, k)$
<i>trans</i> = 'T'	$\max(1, k)$	$\max(1, n)$

beta Specifies the scalar *beta*.

c Array, size $(n * (n + 1) / 2)$. Before entry contains the symmetric matrix *C* in [RFP format](#).

Output Parameters

c If *trans* = 'N' or 'n', then *c* contains $C := \alpha * A * A' + \beta * C$;
if *trans* = 'T' or 't', then *c* contains $C := \alpha * A' * A + \beta * C$;

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?hfrk

Performs a Hermitian rank-*k* operation for matrix in RFP format.

Syntax

```
lapack_int LAPACKE_chfrk( int matrix_layout, char transr, char uplo, char trans,
lapack_int n, lapack_int k, float alpha, const lapack_complex_float* a, lapack_int lda,
float beta, lapack_complex_float* c );
```

```
lapack_int LAPACKE_zhfrk( int matrix_layout, char transr, char uplo, char trans,
lapack_int n, lapack_int k, double alpha, const lapack_complex_double* a, lapack_int
lda, double beta, lapack_complex_double* c );
```

Include Files

- mkl.h

Description

The ?hfrk routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

$$C := \alpha A A^H + \beta C,$$

or

$$C := \alpha A^H A + \beta C,$$

where:

α and β are real scalars,

C is an n -by- n Hermitian matrix in [RFP format](#),

A is an n -by- k matrix in the first case and a k -by- n matrix in the second case.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>transr</i>	if <i>transr</i> = 'N' or 'n', the normal form of RFP C is stored; if <i>transr</i> = 'C' or 'c', the conjugate-transpose form of RFP C is stored.
<i>uplo</i>	Specifies whether the upper or lower triangular part of the array c is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array c is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array c is used.
<i>trans</i>	Specifies the operation: if <i>trans</i> = 'N' or 'n', then $C := \alpha A A^H + \beta C$; if <i>trans</i> = 'C' or 'c', then $C := \alpha A^H A + \beta C$.
<i>n</i>	Specifies the order of the matrix C . The value of n must be at least zero.
<i>k</i>	On entry with <i>trans</i> = 'N' or 'n', k specifies the number of columns of the matrix a , and on entry with <i>trans</i> = 'T' or 't' or 'C' or 'c', k specifies the number of rows of the matrix a . The value of k must be at least zero.
<i>alpha</i>	Specifies the scalar α .
<i>a</i>	Array, size $\max(1, lda * ka)$, where ka is in the following table:

	Col_major	Row_major
<i>trans</i> = 'N'	k	n

<i>trans</i> = 'T'	<i>n</i>	<i>k</i>
--------------------	----------	----------

Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda

Specifies the leading dimension of *a* as declared in the calling (sub)program. *lda* is defined by the following table:

	Col_major	Row_major
<i>trans</i> = 'N'	max(1, <i>n</i>)	max(1, <i>k</i>)
<i>trans</i> = 'T'	max(1, <i>k</i>)	max(1, <i>n</i>)

beta

Specifies the scalar *beta*.

c

Array, size $(n * (n+1) / 2)$. Before entry contains the Hermitian matrix *C* in [RFP format](#).

Output Parameters

c

If *trans* = 'N' or 'n', then *c* contains $C := \alpha * A * A^H + \beta * C$;
if *trans* = 'C' or 'c', then *c* contains $C := \alpha * A^H * A + \beta * C$;

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?tfsm

Solves a matrix equation (one operand is a triangular matrix in RFP format).

Syntax

```
lapack_int LAPACKE_stfsm (int matrix_layout , char transr , char side , char uplo ,
char trans , char diag , lapack_int m , lapack_int n , float alpha , const float * a ,
float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_dtfsm (int matrix_layout , char transr , char side , char uplo ,
char trans , char diag , lapack_int m , lapack_int n , double alpha , const double * a ,
double * b , lapack_int ldb );
```

```
lapack_int LAPACKE_ctfsm (int matrix_layout , char transr , char side , char uplo ,
char trans , char diag , lapack_int m , lapack_int n , lapack_complex_float alpha ,
const lapack_complex_float * a , lapack_complex_float * b , lapack_int ldb );
```

```
lapack_int LAPACKE_ztfsm (int matrix_layout , char transr , char side , char uplo ,
char trans , char diag , lapack_int m , lapack_int n , lapack_complex_double alpha ,
const lapack_complex_double * a , lapack_complex_double * b , lapack_int ldb );
```

Include Files

- `mkl.h`

Description

The `?tfsm` routines solve one of the following matrix equations:

$$\text{op}(A) * X = \alpha * B,$$

or

$$X * \text{op}(A) = \alpha * B,$$

where:

α is a scalar,

X and B are m -by- n matrices,

A is a unit, or non-unit, upper or lower triangular matrix in [rectangular full packed \(RFP\) format](#).

$\text{op}(A)$ can be one of the following:

- $\text{op}(A) = A$ or $\text{op}(A) = A^T$ for real flavors
- $\text{op}(A) = A$ or $\text{op}(A) = A^H$ for complex flavors

The matrix B is overwritten by the solution matrix X .

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>transr</code>	<p>if <code>transr = 'N' or 'n'</code>, the normal form of RFP A is stored;</p> <p>if <code>transr = 'T' or 't'</code>, the transpose form of RFP A is stored;</p> <p>if <code>transr = 'C' or 'c'</code>, the conjugate-transpose form of RFP A is stored.</p>
<code>side</code>	<p>Specifies whether $\text{op}(A)$ appears on the left or right of X in the equation:</p> <p>if <code>side = 'L' or 'l'</code>, then $\text{op}(A) * X = \alpha * B$;</p> <p>if <code>side = 'R' or 'r'</code>, then $X * \text{op}(A) = \alpha * B$.</p>
<code>uplo</code>	<p>Specifies whether the RFP matrix A is upper or lower triangular:</p> <p>if <code>uplo = 'U' or 'u'</code>, then the matrix is upper triangular;</p> <p>if <code>uplo = 'L' or 'l'</code>, then the matrix is low triangular.</p>
<code>trans</code>	<p>Specifies the form of $\text{op}(A)$ used in the matrix multiplication:</p> <p>if <code>trans = 'N' or 'n'</code>, then $\text{op}(A) = A$;</p> <p>if <code>trans = 'T' or 't'</code>, then $\text{op}(A) = A'$;</p> <p>if <code>trans = 'C' or 'c'</code>, then $\text{op}(A) = \text{conjg}(A')$.</p>
<code>diag</code>	<p>Specifies whether the RFP matrix A is unit triangular:</p> <p>if <code>diag = 'U' or 'u'</code> then the matrix is unit triangular;</p> <p>if <code>diag = 'N' or 'n'</code>, then the matrix is not unit triangular.</p>

<i>m</i>	Specifies the number of rows of <i>B</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	Specifies the number of columns of <i>B</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	Specifies the scalar <i>alpha</i> . When <i>alpha</i> is zero, then <i>a</i> is not referenced and <i>b</i> need not be set before entry.
<i>a</i>	Array, size $(n*(n+1)/2)$. Contains the matrix <i>A</i> in RFP format .
<i>b</i>	Array, size $\max(1, ldb*n)$ for column major and $\max(1, ldb*m)$ for row major. Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the right-hand side matrix <i>B</i> .
<i>ldb</i>	Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. The value of <i>ldb</i> must be at least $\max(1, m)$ for column major and $\max(1, n)$ for row major.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?tfttp

Copies a triangular matrix from the rectangular full packed format (TF) to the standard packed format (TP).

Syntax

```
lapack_int LAPACKE_stfttp (int matrix_layout , char transr , char uplo , lapack_int n ,
const float * arf , float * ap );
```

```
lapack_int LAPACKE_dtfttp (int matrix_layout , char transr , char uplo , lapack_int n ,
const double * arf , double * ap );
```

```
lapack_int LAPACKE_ctfttp (int matrix_layout , char transr , char uplo , lapack_int n ,
const lapack_complex_float * arf , lapack_complex_float * ap );
```

```
lapack_int LAPACKE_ztfttp (int matrix_layout , char transr , char uplo , lapack_int n ,
const lapack_complex_double * arf , lapack_complex_double * ap );
```

Include Files

- mkl.h

Description

The routine copies a triangular matrix *A* from the Rectangular Full Packed (RFP) format to the standard packed format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>transr</i>	= 'N': <i>arf</i> is in the Normal format, = 'T': <i>arf</i> is in the Transpose format (for <i>stfttp</i> and <i>dtfttp</i>), = 'C': <i>arf</i> is in the Conjugate-transpose format (for <i>ctfttp</i> and <i>ztfttp</i>).
<i>uplo</i>	Specifies whether <i>A</i> is upper or lower triangular: = 'U': <i>A</i> is upper triangular, = 'L': <i>A</i> is lower triangular.
<i>n</i>	The order of the matrix <i>A</i> . $n \geq 0$.
<i>arf</i>	Array, size at least $\max(1, n*(n+1)/2)$. On entry, the upper or lower triangular matrix <i>A</i> stored in the RFP format.

Output Parameters

<i>ap</i>	Array, size at least $\max(1, n*(n+1)/2)$. On exit, the upper or lower triangular matrix <i>A</i> , packed columnwise in a linear array.
-----------	--

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?tfttr

Copies a triangular matrix from the rectangular full packed format (TF) to the standard full format (TR) .

Syntax

```
lapack_int LAPACKE_stfttr (int matrix_layout , char transr , char uplo , lapack_int n ,
const float * arf , float * a , lapack_int lda );

lapack_int LAPACKE_dtfttr (int matrix_layout , char transr , char uplo , lapack_int n ,
const double * arf , double * a , lapack_int lda );

lapack_int LAPACKE_ctfttr (int matrix_layout , char transr , char uplo , lapack_int n ,
const lapack_complex_float * arf , lapack_complex_float * a , lapack_int lda );

lapack_int LAPACKE_ztfttr (int matrix_layout , char transr , char uplo , lapack_int n ,
const lapack_complex_double * arf , lapack_complex_double * a , lapack_int lda );
```

Include Files

- `mk1.h`

Description

The routine copies a triangular matrix A from the Rectangular Full Packed (RFP) format to the standard full format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>transr</code>	= 'N': <i>arf</i> is in the Normal format, = 'T': <i>arf</i> is in the Transpose format (for <code>stfttr</code> and <code>dtfttr</code>), = 'C': <i>arf</i> is in the Conjugate-transpose format (for <code>ctfttr</code> and <code>ztfttr</code>).
<code>uplo</code>	Specifies whether A is upper or lower triangular: = 'U': A is upper triangular, = 'L': A is lower triangular.
<code>n</code>	The order of the matrices <i>arf</i> and a . $n \geq 0$.
<code>arf</code>	Array, size at least $\max(1, n*(n+1)/2)$. On entry, the upper or lower triangular matrix A stored in the RFP format.
<code>lda</code>	The leading dimension of the array a . $lda \geq \max(1, n)$.

Output Parameters

<code>a</code>	Array, size $\max(1, lda * n)$. On exit, the triangular matrix A . If <code>uplo</code> = 'U', the leading n -by- n upper triangular part of the array a contains the upper triangular matrix, and the strictly lower triangular part of a is not referenced. If <code>uplo</code> = 'L', the leading n -by- n lower triangular part of the array a contains the lower triangular matrix, and the strictly upper triangular part of a is not referenced.
----------------	--

Return Values

This function returns a value *info*.

If `info` = 0, the execution is successful.

If `info` < 0, the i -th parameter had an illegal value.

If `info` = -1011, memory allocation error occurred.

?tpqrt2

Computes a QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for Q.

Syntax

```
lapack_int LAPACKE_stpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, float * a, lapack_int lda, float * b, lapack_int ldb, float * t, lapack_int ldt);
lapack_int LAPACKE_dtpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, double * a, lapack_int lda, double * b, lapack_int ldb, double * t, lapack_int ldt);
lapack_int LAPACKE_ctpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb, lapack_complex_float * t, lapack_int ldt );
lapack_int LAPACKE_ztpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb, lapack_complex_double * t, lapack_int ldt );
```

Include Files

- mkl.h

Description

The input matrix C is an $(n+m)$ -by- n matrix

$$C = \begin{bmatrix} A \\ B \end{bmatrix} \begin{matrix} \leftarrow n \times n \text{ upper triangular} \\ \leftarrow m \times n \text{ pentagonal} \end{matrix}$$

where A is an n -by- n upper triangular matrix, and B is an m -by- n pentagonal matrix consisting of an $(m-1)$ -by- n rectangular matrix $B1$ on top of an 1 -by- n upper trapezoidal matrix $B2$:

$$B = \begin{bmatrix} B1 \\ B2 \end{bmatrix} \begin{matrix} \leftarrow (m-1) \times n \text{ rectangular} \\ \leftarrow 1 \times n \text{ upper trapezoidal} \end{matrix}$$

The upper trapezoidal matrix $B2$ consists of the first l rows of an n -by- n upper triangular matrix, where $0 \leq l \leq \min(m, n)$. If $l=0$, B is an m -by- n rectangular matrix. If $m=l=n$, B is upper triangular. The matrix W contains the elementary reflectors $H(i)$ in the i th column below the diagonal (of A) in the $(n+m)$ -by- n input matrix C so that W can be represented as

$$W = \begin{bmatrix} I \\ V \end{bmatrix} \begin{matrix} \leftarrow n \times n \text{ identity} \\ \leftarrow m \times n \text{ pentagonal} \end{matrix}$$

Thus, V contains all of the information needed for W , and is returned in array b .

NOTE

V has the same form as B :

$$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix} \begin{matrix} \leftarrow (m-l) \times n \text{ rectangular} \\ \leftarrow l \times n \text{ upper trapezoidal} \end{matrix}$$

The columns of V represent the vectors which define the $H(i)$ s.

The $(m+n)$ -by- $(m+n)$ block reflector H is then given by

$H = I - W^* T^* W^T$ for real flavors, and

$H = I - W^* T^* W^H$ for complex flavors

where W^T is the transpose of W , W^H is the conjugate transpose of W , and T is the upper triangular factor of the block reflector.

Input Parameters

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<code>m</code>	The total number of rows in the matrix B ($m \geq 0$).
<code>n</code>	The number of columns in B and the order of the triangular matrix A ($n \geq 0$).
<code>l</code>	The number of rows of the upper trapezoidal part of B ($\min(m, n) \geq l \geq 0$).
<code>a, b</code>	Arrays: a , size $\max(1, lda * n)$ contains the n -by- n upper triangular matrix A . b , size $\max(1, ldb * n)$ for column major and $\max(1, ldb * m)$ for row major, the pentagonal m -by- n matrix B . The first $(m-l)$ rows contain the rectangular $B1$ matrix, and the next l rows contain the upper trapezoidal $B2$ matrix.
<code>lda</code>	The leading dimension of a ; at least $\max(1, n)$.
<code>ldb</code>	The leading dimension of b ; at least $\max(1, m)$ for column major and $\max(1, n)$ for row major.

ldt The leading dimension of *t*; at least $\max(1, n)$.

Output Parameters

a The elements on and above the diagonal of the array contain the upper triangular matrix *R*.

b The pentagonal matrix *V*.

t Array, size $\max(1, ldt * n)$.

The upper *n*-by-*n* upper triangular factor *T* of the block reflector.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0 and *info* = -*i*, the *i*th argument had an illegal value.

If *info* = -1011, memory allocation error occurred.

?tprfb

Applies a real or complex "triangular-pentagonal" blocked reflector to a real or complex matrix, which is composed of two blocks.

Syntax

```
lapack_int LAPACKE_stprfb (int matrix_layout, char side, char trans, char direct, char storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const float * v, lapack_int ldv, const float * t, lapack_int ldt, float * a, lapack_int lda, float * b, lapack_int ldb);
```

```
lapack_int LAPACKE_dtprfb (int matrix_layout, char side, char trans, char direct, char storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const double * v, lapack_int ldv, const double * t, lapack_int ldt, double * a, lapack_int lda, double * b, lapack_int ldb);
```

```
lapack_int LAPACKE_ctprfb (int matrix_layout, char side, char trans, char direct, char storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const lapack_complex_float * v, lapack_int ldv, const lapack_complex_float * t, lapack_int ldt, lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb);
```

```
lapack_int LAPACKE_ztprfb (int matrix_layout, char side, char trans, char direct, char storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const lapack_complex_double * v, lapack_int ldv, const lapack_complex_double * t, lapack_int ldt, lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb);
```

Include Files

- mkl.h

Description

The `?tprfb` routine applies a real or complex "triangular-pentagonal" block reflector H , H^T , or H^H from either the left or the right to a real or complex matrix C , which is composed of two blocks A and B .

The block B is m -by- n . If `side = 'R'`, A is m -by- k , and if `side = 'L'`, A is of size k -by- n .

$$\begin{array}{ll}
 \text{direct} = \text{'F'} & \text{direct} = \text{'B'} \\
 \text{side} = \text{'R'} & C = \begin{bmatrix} A & B \end{bmatrix} \quad C = \begin{bmatrix} B & A \end{bmatrix} \\
 \text{side} = \text{'L'} & C = \begin{bmatrix} A \\ B \end{bmatrix} \quad C = \begin{bmatrix} B \\ A \end{bmatrix}
 \end{array}$$

The pentagonal matrix V is composed of a rectangular block $V1$ and a trapezoidal block $V2$. The size of the trapezoidal block is determined by the parameter l , where $0 \leq l \leq k$. if $l=k$, the $V2$ block of V is triangular; if $l=0$, there is no trapezoidal block, thus $V = V1$ is rectangular.

	<code>direct='F'</code>	<code>direct='B'</code>
<code>storev='C'</code>	$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix}$ <p>$V2$ is upper trapezoidal (first l rows of k-by-k upper triangular)</p>	$V = \begin{bmatrix} V2 \\ V1 \end{bmatrix}$ <p>$V2$ is lower trapezoidal (last l rows of k-by-k lower triangular matrix)</p>
<code>storev='R'</code>	$V = \begin{bmatrix} V1 & V2 \end{bmatrix}$ <p>$V2$ is lower trapezoidal (first l columns of k-by-k lower triangular matrix)</p>	$V = \begin{bmatrix} V2 & V1 \end{bmatrix}$ <p>$V2$ is upper trapezoidal (last l columns of k-by-k upper triangular matrix)</p>

	<code>side='L'</code>	<code>side='R'</code>
<code>storev='C'</code>	V is m -by- k $V2$ is l -by- k	V is n -by- k $V2$ is l -by- k
<code>storev='R'</code>	V is k -by- m $V2$ is k -by- l	V is k -by- n $V2$ is k -by- l

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>side</i>	= 'L': apply H , H^T , or H^H from the left, = 'R': apply H , H^T , or H^H from the right.
<i>trans</i>	= 'N': apply H (no transpose), = 'T': apply H^T (transpose), = 'C': apply H^H (conjugate transpose).
<i>direct</i>	Indicates how H is formed from a product of elementary reflectors: = 'F': $H = H(1) H(2) \dots H(k)$ (Forward), = 'B': $H = H(k) \dots H(2) H(1)$ (Backward).
<i>storev</i>	Indicates how the vectors that define the elementary reflectors are stored: = 'C': Columns, = 'R': Rows.
<i>m</i>	The total number of rows in the matrix B ($m \geq 0$).
<i>n</i>	The number of columns in B ($n \geq 0$).
<i>k</i>	The order of the matrix T , which is the number of elementary reflectors whose product defines the block reflector. ($k \geq 0$)
<i>l</i>	The order of the trapezoidal part of V . ($k \geq l \geq 0$).
<i>v</i>	An array containing the pentagonal matrix V (the elementary reflectors $H(1)$, $H(2)$, ..., $H(k)$). The size limitations depend on values of parameters <i>storev</i> and <i>side</i> as described in the following table

	<i>storev</i> = C		<i>storev</i> = R	
	<i>side</i> = L	<i>side</i> = R	<i>side</i> = L	<i>side</i> = R
Column major	$\max(1, l_{dv} * k)$	$\max(1, l_{dv} * k)$	$\max(1, l_{dv} * m)$	$\max(1, l_{dv} * n)$
Row major	$\max(1, l_{dv} * m)$	$\max(1, l_{dv} * n)$	$\max(1, l_{dv} * k)$	$\max(1, l_{dv} * k)$

ldv The leading dimension of the array *v*. It should satisfy the following conditions:

	<i>storev</i> = C		<i>storev</i> = R	
	<i>side</i> = L	<i>side</i> = R	<i>side</i> = L	<i>side</i> = R
Column major	$\max(1, m)$	$\max(1, n)$	$\max(1, k)$	$\max(1, k)$
Row major	$\max(1, k)$	$\max(1, k)$	$\max(1, m)$	$\max(1, n)$

t Array size $\max(1, ldt * k)$. The triangular k -by- k matrix T in the representation of the block reflector.

ldt The leading dimension of the array t ($ldt \geq k$).

a size should satisfy the following conditions:

k if $side = 'R'$.

	$side = L$	$side = R$
Column major	$\max(1, lda * n)$	$\max(1, lda * k)$
Row major	$\max(1, lda * k)$	$\max(1, lda * m)$

The k -by- n or m -by- k matrix A .

lda The leading dimension of the array a should satisfy the following conditions:

	$side = L$	$side = R$
Column major	$\max(1, k)$	$\max(1, m)$
Row major	$\max(1, n)$	$\max(1, k)$

b Array size at least $\max(1, ldb * n)$ for column major layout and $\max(1, ldb * m)$ for row major layout, the m -by- n matrix B .

ldb The leading dimension of the array b ($ldb \geq \max(1, m)$ for column major layout and $ldb \geq \max(1, n)$ for row major layout).

Output Parameters

a Contains the corresponding block of H^*C , H^T*C , H^H*C , C^*H , C^*H^T , or C^*H^H .

b Contains the corresponding block of H^*C , H^T*C , H^H*C , C^*H , C^*H^T , or C^*H^H .

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?tpddf

Copies a triangular matrix from the standard packed format (TP) to the rectangular full packed format (TF).

Syntax

```
lapack_int LAPACKE_stpddf (int matrix_layout , char transr , char uplo , lapack_int n ,
const float * ap , float * arf );
```

```
lapack_int LAPACKE_dtpddf (int matrix_layout , char transr , char uplo , lapack_int n ,
const double * ap , double * arf );
```

```
lapack_int LAPACKE_ctpddf (int matrix_layout , char transr , char uplo , lapack_int n ,
const lapack_complex_float * ap , lapack_complex_float * arf );
```

```
lapack_int LAPACKE_ztpddf (int matrix_layout , char transr , char uplo , lapack_int n ,
const lapack_complex_double * ap , lapack_complex_double * arf );
```

Include Files

- mkl.h

Description

The routine copies a triangular matrix *A* from the standard packed format to the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>transr</i>	= 'N': <i>arf</i> must be in the Normal format, = 'T': <i>arf</i> must be in the Transpose format (for <i>stpttf</i> and <i>dtpttf</i>), = 'C': <i>arf</i> must be in the Conjugate-transpose format (for <i>ctpttf</i> and <i>ztpttf</i>).
<i>uplo</i>	Specifies whether <i>A</i> is upper or lower triangular: = 'U': <i>A</i> is upper triangular, = 'L': <i>A</i> is lower triangular.
<i>n</i>	The order of the matrix <i>A</i> . $n \geq 0$.
<i>ap</i>	Array, size at least $\max(1, n*(n+1)/2)$. On entry, the upper or lower triangular matrix <i>A</i> , packed in a linear array. See Matrix Storage Schemes for more information.

Output Parameters

<i>arf</i>	Array, size at least $\max(1, n*(n+1)/2)$. On exit, the upper or lower triangular matrix <i>A</i> stored in the RFP format.
------------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

< 0: if *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?tptr

Copies a triangular matrix from the standard packed format (TP) to the standard full format (TR) .

Syntax

```
lapack_int LAPACKE_stpttr (int matrix_layout , char uplo , lapack_int n , const float *
ap , float * a , lapack_int lda );

lapack_int LAPACKE_dtptr (int matrix_layout , char uplo , lapack_int n , const double
* ap , double * a , lapack_int lda );

lapack_int LAPACKE_ctpttr (int matrix_layout , char uplo , lapack_int n , const
lapack_complex_float * ap , lapack_complex_float * a , lapack_int lda );

lapack_int LAPACKE_ztptr (int matrix_layout , char uplo , lapack_int n , const
lapack_complex_double * ap , lapack_complex_double * a , lapack_int lda );
```

Include Files

- mkl.h

Description

The routine copies a triangular matrix *A* from the standard packed format to the standard full format.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies whether <i>A</i> is upper or lower triangular: = 'U': <i>A</i> is upper triangular, = 'L': <i>A</i> is lower triangular.
<i>n</i>	The order of the matrices <i>ap</i> and <i>a</i> . $n \geq 0$.
<i>ap</i>	Array, size at least $\max(1, n*(n+1)/2)$. (see Matrix Storage Schemes).
<i>lda</i>	The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.

Output Parameters

<i>a</i>	Array, size $\max(1, lda*n)$. On exit, the triangular matrix <i>A</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>a</i> is not referenced.
----------	---

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?trttf

Copies a triangular matrix from the standard full format (TR) to the rectangular full packed format (TF).

Syntax

```
lapack_int LAPACKE_strttf (int matrix_layout , char transr , char uplo , lapack_int n ,
const float * a , lapack_int lda , float * arf );
```

```
lapack_int LAPACKE_dtrttf (int matrix_layout , char transr , char uplo , lapack_int n ,
const double * a , lapack_int lda , double * arf );
```

```
lapack_int LAPACKE_ctrttf (int matrix_layout , char transr , char uplo , lapack_int n ,
const lapack_complex_float * a , lapack_int lda , lapack_complex_float * arf );
```

```
lapack_int LAPACKE_ztrttf (int matrix_layout , char transr , char uplo , lapack_int n ,
const lapack_complex_double * a , lapack_int lda , lapack_complex_double * arf );
```

Include Files

- mkl.h

Description

The routine copies a triangular matrix *A* from the standard full format to the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>transr</i>	= 'N': <i>arf</i> must be in the Normal format, = 'T': <i>arf</i> must be in the Transpose format (for <i>strttf</i> and <i>dtrttf</i>), = 'C': <i>arf</i> must be in the Conjugate-transpose format (for <i>ctrttf</i> and <i>ztrttf</i>).
<i>uplo</i>	Specifies whether <i>A</i> is upper or lower triangular: = 'U': <i>A</i> is upper triangular, = 'L': <i>A</i> is lower triangular.
<i>n</i>	The order of the matrix <i>A</i> . $n \geq 0$.
<i>a</i>	Array, size $\max(1, (lda * n))$. On entry, the triangular matrix <i>A</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> contains the upper triangular matrix, and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> contains the lower triangular matrix, and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.

Output Parameters

arf Array, size at least $\max(1, n*(n+1)/2)$.
On exit, the upper or lower triangular matrix *A* stored in the RFP format.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?trttp

Copies a triangular matrix from the standard full format (TR) to the standard packed format (TP) .

Syntax

```
lapack_int LAPACKE_strttp (int matrix_layout , char uplo , lapack_int n , const float *
a , lapack_int lda , float * ap );
```

```
lapack_int LAPACKE_dtrttp (int matrix_layout , char uplo , lapack_int n , const double
* a , lapack_int lda , double * ap );
```

```
lapack_int LAPACKE_ctrttp (int matrix_layout , char uplo , lapack_int n , const
lapack_complex_float * a , lapack_int lda , lapack_complex_float * ap );
```

```
lapack_int LAPACKE_ztrttp (int matrix_layout , char uplo , lapack_int n , const
lapack_complex_double * a , lapack_int lda , lapack_complex_double * ap );
```

Include Files

- mkl.h

Description

The routine copies a triangular matrix *A* from the standard full format to the standard packed format.

Input Parameters

uplo Specifies whether *A* is upper or lower triangular:
= 'U': *A* is upper triangular,
= 'L': *A* is lower triangular.

n The order of the matrix *A*, $n \geq 0$.

a Array, size $\max(1, lda * n)$.
On entry, the triangular matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced.

lda The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

ap Array, size at least $\max(1, n*(n+1)/2)$.
On exit, the upper or lower triangular matrix *A*, packed columnwise in a linear array. (see [Matrix Storage Schemes](#))

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

?lapc2

Copies all or part of a real two-dimensional array to a complex array.

Syntax

```
lapack_int LAPACKE_clapc2 (int matrix_layout , char uplo , lapack_int m , lapack_int
n , const float * a , lapack_int lda , lapack_complex_float * b , lapack_int ldb );
lapack_int LAPACKE_zlapc2 (int matrix_layout , char uplo , lapack_int m , lapack_int
n , const double * a , lapack_int lda , lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine copies all or part of a real matrix *A* to another matrix *B*.

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo Specifies the part of the matrix *A* to be copied to *B*.
If *uplo* = 'U', the upper triangular part of *A*;
if *uplo* = 'L', the lower triangular part of *A*.
Otherwise, all of the matrix *A* is copied.

m The number of rows in the matrix *A* ($m \geq 0$).

n The number of columns in *A* ($n \geq 0$).

a Array, size at least $\max(1, lda*n)$ for column major and $\max(1, lda*m)$ for row major, contains the *m*-by-*n* matrix *A*.

If `uplo = 'U'`, only the upper triangle or trapezoid is accessed; if `uplo = 'L'`, only the lower triangle or trapezoid is accessed.

`lda` The leading dimension of `a`; $lda \geq \max(1, m)$ for column major and $lda \geq \max(1, n)$ for row major.

`ldb` The leading dimension of the output array `b`; $ldb \geq \max(1, m)$ for column major and $ldb \geq \max(1, n)$ for row major.

Output Parameters

`b` Array, size at least $\max(1, ldb \cdot n)$ for column major layout and $\max(1, ldb \cdot m)$ for row major layout, contains the m -by- n matrix B .
On exit, $B = A$ in the locations specified by `uplo`.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info < 0`, the i -th parameter had an illegal value.

If `info = -1011`, memory allocation error occurred.

?larcm

Multiplies a square real matrix by a complex matrix.

Syntax

```
lapack_int LAPACKC_clarcm(int matrix_layout, lapack_int m, lapack_int n, const float
*a, lapack_int lda, const lapack_complex_float * b, lapack_int ldb, lapack_complex_float *
c, lapack_int ldc);
```

```
lapack_int LAPACKC_zlarcm(int matrix_layout, lapack_int m, lapack_int n, const double *
a, lapack_int lda, const lapack_complex_double *b, lapack_int ldb, lapack_complex_double
*c , lapack_int ldc);
```

Description

The routine performs a simple matrix-matrix multiplication of the form

$$C = A * B,$$

where A is m -by- m and real, B is m -by- n and complex, and C is m -by- n and complex.

Input Parameters

`m` The number of rows and columns of matrix A and the number of rows of matrix C ($m \geq 0$).

`n` The number of columns of matrix B and the number of columns of matrix C ($n \geq 0$).

`a` Array, size $[lda * m]$. Contains the m -by- m matrix A .

<i>lda</i>	The leading dimension of the array <i>a</i> , $lda \geq \max(1, m)$.
<i>b</i>	Array, size (<i>ldb</i> , <i>n</i>). Contains the <i>m</i> -by- <i>n</i> matrix <i>B</i> .
<i>ldb</i>	The leading dimension of the array <i>b</i> , $ldb \geq \max(1, m)$ for column-major layout; $ldb \geq \max(1, n)$ for row-major layout.
<i>ldc</i>	The leading dimension of the array <i>c</i> , $ldc \geq \max(1, m)$ for column-major layout; $ldc \geq \max(1, n)$ for row-major layout.

Output Parameters

<i>c</i>	Array, size (<i>ldc</i> , <i>n</i>). Contains the <i>m</i> -by- <i>n</i> matrix <i>C</i> .
----------	--

Return Values

This function returns a value *info*. If *info* = 0, the execution is successful. If *info* = -*i*, parameter *i* had an illegal value.

mkl_?tppack

Copies a triangular/symmetric matrix or submatrix from standard full format to standard packed format.

Syntax

```
lapack_int LAPACKE_mkl_stppack (int matrix_layout, char uplo, char trans, lapack_int n,
float* ap, lapack_int i, lapack_int j, lapack_int rows, lapack_int cols, const float* a,
lapack_int lda);
```

```
lapack_int LAPACKE_mkl_dtpack (int matrix_layout, char uplo, char trans, lapack_int n,
double* ap, lapack_int i, lapack_int j, lapack_int rows, lapack_int cols, const double*
a, lapack_int lda);
```

```
lapack_int LAPACKE_mkl_ctppack (int matrix_layout, char uplo, char trans, lapack_int n,
MKL_Complex8* ap, lapack_int i, lapack_int j, lapack_int rows, lapack_int cols, const
MKL_Complex8* a, lapack_int lda);
```

```
lapack_int LAPACKE_mkl_ztpack (int matrix_layout, char uplo, char trans, lapack_int n,
MKL_Complex16* ap, lapack_int i, lapack_int j, lapack_int rows, lapack_int cols, const
MKL_Complex16* a, lapack_int lda);
```

Include Files

- mkl.h

Description

The routine copies a triangular or symmetric matrix or its submatrix from standard full format to packed format

```
 $AP_{i:i+rows-1, j:j+cols-1} := op(A)$ 
```

Standard packed formats include:

- TP: triangular packed storage
- SP: symmetric indefinite packed storage
- HP: Hermitian indefinite packed storage
- PP: symmetric or Hermitian positive definite packed storage

Full formats include:

- GE: general
- TR: triangular
- SY: symmetric indefinite
- HE: Hermitian indefinite
- PO: symmetric or Hermitian positive definite

NOTE

Any elements of the copied submatrix rectangular outside of the triangular part of the matrix AP are skipped.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>uplo</i>	Specifies whether the matrix AP is upper or lower triangular. If <i>uplo</i> = 'U', AP is upper triangular. If <i>uplo</i> = 'L': AP is lower triangular.
<i>trans</i>	Specifies whether or not the copied block of A is transposed or not. If <i>trans</i> = 'N', no transpose: $\text{op}(A) = A$. If <i>trans</i> = 'T', transpose: $\text{op}(A) = A^T$. If <i>trans</i> = 'C', conjugate transpose: $\text{op}(A) = A^H$. For real data this is the same as <i>trans</i> = 'T'.
<i>n</i>	The order of the matrix AP ; $n \geq 0$
<i>i, j</i>	Coordinates of the left upper corner of the destination submatrix in AP . If <i>uplo</i> ='U', $1 \leq i \leq j \leq n$. If <i>uplo</i> ='L', $1 \leq j \leq i \leq n$.
<i>rows</i>	Number of rows in the destination submatrix. $0 \leq \text{rows} \leq n - i + 1$.
<i>cols</i>	Number of columns in the destination submatrix. $0 \leq \text{cols} \leq n - j + 1$.
<i>a</i>	Pointer to the source submatrix. Array <i>a</i> contains the <i>rows</i> -by- <i>cols</i> submatrix stored as unpacked rows-by-columns if <i>trans</i> = 'N', or unpacked columns-by-rows if <i>trans</i> = 'T' or <i>trans</i> = 'C'. The size of <i>a</i> is

	<i>trans</i> = 'N'	<i>trans</i> ='T' or <i>trans</i> ='C'
<i>matrix_layout</i> = LAPACK_COL_MAJOR	<i>lda</i> * <i>cols</i>	<i>lda</i> * <i>rows</i>
<i>matrix_layout</i> = LAPACK_ROW_MAJOR	<i>lda</i> * <i>rows</i>	<i>lda</i> * <i>cols</i>

NOTE

If there are elements outside of the triangular part of AP , they are skipped and are not copied from a .

lda

The leading dimension of the array a .

	<i>trans</i> = 'N'	<i>trans</i> ='T' or <i>trans</i> ='C'
<i>matrix_layout</i> = LAPACK_COL_MAJOR	<i>lda</i> ≥ max(1, <i>rows</i>)	<i>lda</i> ≥ max(1, <i>cols</i>)
<i>matrix_layout</i> = LAPACK_ROW_MAJOR	<i>lda</i> ≥ max(1, <i>cols</i>)	<i>lda</i> ≥ max(1, <i>rows</i>)

Output Parameters

ap

Array of size at least $\max(1, n(n+1)/2)$. The array ap contains either the upper or the lower triangular part of the matrix AP (as specified by *uplo*) in packed storage (see [Matrix Storage Schemes](#)). The submatrix of ap from row i to row $i + rows - 1$ and column j to column $j + cols - 1$ is overwritten with a copy of the source matrix.

Return Values

This function returns a value *info*. If *info*=0, the execution is successful. If *info* = $-i$, the i -th parameter had an illegal value.

mkl_?tpunpack

Copies a triangular/symmetric matrix or submatrix from standard packed format to full format.

Syntax

```
lapack_int LAPACKE_mkl_stpunpack ( int matrix_layout, char uplo, char trans,
    lapack_int n, const float* ap, lapack_int i, lapack_int j, lapack_int rows,
    lapack_int cols, float* a, lapack_int lda );
```

```
lapack_int LAPACKE_mkl_dtpunpack ( int matrix_layout, char uplo, char trans,
    lapack_int n, const double* ap, lapack_int i, lapack_int j, lapack_int rows,
    lapack_int cols, double* a, lapack_int lda );
```

```
lapack_int LAPACKE_mkl_ctpunpack ( int matrix_layout, char uplo, char trans,
    lapack_int n, const MKL_Complex8* ap, lapack_int i, lapack_int j, lapack_int rows,
    lapack_int cols, MKL_Complex8* a, lapack_int lda );
```

```
lapack_int LAPACKE_mkl_ztpunpack ( int matrix_layout, char uplo, char trans,
    lapack_int n, const MKL_Complex16* ap, lapack_int i, lapack_int j, lapack_int rows,
    lapack_int cols, MKL_Complex16* a, lapack_int lda );
```

Include Files

- mkl.h

Description

The routine copies a triangular or symmetric matrix or its submatrix from standard packed format to full format.

```
A := op(APi:i+rows-1, j:j+cols-1)
```

Standard packed formats include:

- TP: triangular packed storage
- SP: symmetric indefinite packed storage
- HP: Hermitian indefinite packed storage
- PP: symmetric or Hermitian positive definite packed storage

Full formats include:

- GE: general
- TR: triangular
- SY: symmetric indefinite
- HE: Hermitian indefinite
- PO: symmetric or Hermitian positive definite

NOTE

Any elements of the copied submatrix rectangular outside of the triangular part of *AP* are skipped.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
<i>uplo</i>	Specifies whether matrix <i>AP</i> is upper or lower triangular. If <i>uplo</i> = 'U', <i>AP</i> is upper triangular. If <i>uplo</i> = 'L': <i>AP</i> is lower triangular.
<i>trans</i>	Specifies whether or not the copied block of <i>AP</i> is transposed. If <i>trans</i> = 'N', no transpose: $\text{op}(AP) = AP$. If <i>trans</i> = 'T', transpose: $\text{op}(AP) = AP^T$. If <i>trans</i> = 'C', conjugate transpose: $\text{op}(AP) = AP^H$. For real data this is the same as <i>trans</i> = 'T'.
<i>n</i>	The order of the matrix <i>AP</i> ; $n \geq 0$.
<i>ap</i>	Array, size at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains either the upper or the lower triangular part of the matrix <i>AP</i> (as specified by <i>uplo</i>) in packed storage (see Matrix Storage Schemes). It is the source for the submatrix of <i>AP</i> from row <i>i</i> to row <i>i</i> + rows - 1 and column <i>j</i> to column <i>j</i> + cols - 1 to be copied.
<i>i, j</i>	Coordinates of left upper corner of the submatrix in <i>AP</i> to copy. If <i>uplo</i> ='U', $1 \leq i \leq j \leq n$.

If $uplo='L'$, $1 \leq j \leq i \leq n$.

rows

Number of rows to copy. $0 \leq rows \leq n - i + 1$.

cols

Number of columns to copy. $0 \leq cols \leq n - j + 1$.

lda

The leading dimension of array *a*.

	<i>trans</i> = 'N'	<i>trans</i> ='T' or <i>trans</i> ='C'
<i>matrix_layout</i> = LAPACK_COL_MAJOR	$lda \geq \max(1, rows)$	$lda \geq \max(1, cols)$
<i>matrix_layout</i> = LAPACK_ROW_MAJOR	$lda \geq \max(1, cols)$	$lda \geq \max(1, rows)$

Output Parameters

a

Pointer to the destination matrix. On exit, array *a* is overwritten with a copy of the unpacked *rows*-by-*cols* submatrix of *ap* unpacked rows-by-columns if *trans* = 'N', or unpacked columns-by-rows if *trans* = 'T' or *trans* = 'C'.

The size of *a* is

	<i>trans</i> = 'N'	<i>trans</i> ='T' or <i>trans</i> ='C'
<i>matrix_layout</i> = LAPACK_COL_MAJOR	$lda * cols$	$lda * rows$
<i>matrix_layout</i> = LAPACK_ROW_MAJOR	$lda * rows$	$lda * cols$

NOTE

If there are elements outside of the triangular part of *ap* indicated by *uplo*, they are skipped and are not copied to *a*.

Return Values

This function returns a value *info*. If *info*=0, the execution is successful. If *info* = -*i*, the *i*-th parameter had an illegal value.

LAPACK Utility Functions and Routines

This section describes LAPACK utility functions and routines.

Summary information about these routines is given in the following table:

LAPACK Utility Routines

Routine Name	Data Types	Description
<code>ilaver</code>		Returns the version of the Lapack library.
<code>ilaenv</code>		Environmental enquiry function which returns values for tuning algorithmic performance.
<code>?lamch</code>	s, d	Determines machine parameters for floating-point arithmetic.

See Also

`lsame` Tests two characters for equality regardless of the case.

`lsamen` Tests two character strings for equality regardless of the case.

`second/dsecnd` Returns elapsed time in seconds. Use to estimate real time between two calls to this function.

`xerbla` Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.

ilaver

Returns the version of the LAPACK library.

Syntax

```
void LAPACKE_ilaver (lapack_int * vers_major, lapack_int * vers_minor, lapack_int *
vers_patch);
```

Include Files

- `mkl.h`

Description

This routine returns the version of the LAPACK library.

Output Parameters

<code>vers_major</code>	Returns the major version of the LAPACK library.
<code>vers_minor</code>	Returns the minor version from the major version of the LAPACK library.
<code>vers_patch</code>	Returns the patch version from the minor version of the LAPACK library.

ilaenv

Environmental enquiry function that returns values for tuning algorithmic performance.

Syntax

```
MKL_INT ilaenv (const MKL_INT *ispec, const char *name, const char *opts, const MKL_INT
*n1, const MKL_INT *n2, const MKL_INT *n3, const MKL_INT *n4);
```

Include Files

- `mkl.h`

Description

The enquiry function `ilaenv` is called from the LAPACK routines to choose problem-dependent parameters for the local environment. See *ispec* below for a description of the parameters.

This version provides a set of parameters that should give good, but not optimal, performance on many of the currently available computers.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

ispec

Specifies the parameter to be returned as the value of `ilaenv`:

- = 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance.
- = 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used.
- = 3: the crossover point (in a block routine, for n less than this value, an unblocked routine should be used)
- = 4: the number of shifts, used in the nonsymmetric eigenvalue routines (deprecated)
- = 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least k -by- m , where k is given by `ilaenv(2, ...)` and m by `ilaenv(5, ...)`
- = 6: the crossover point for the SVD (when reducing an m -by- n matrix to bidiagonal form, if $\max(m, n) / \min(m, n)$ exceeds this value, a QR factorization is used first to reduce the matrix to a triangular form.)
- = 7: the number of processors
- = 8: the crossover point for the multishift QR and QZ methods for nonsymmetric eigenvalue problems (deprecated).
- = 9: maximum size of the subproblems at the bottom of the computation tree in the divide-and-conquer algorithm (used by `?gelsd` and `?gesdd`)
- =10: ieee NaN arithmetic can be trusted not to trap
- =11: infinity arithmetic can be trusted not to trap
- $12 \leq ispec \leq 16$: `?hseqr` or one of its subroutines, see `iparmq` for detailed explanation.

name

The name of the calling subroutine, in either upper case or lower case.

opts

The character options to the subroutine *name*, concatenated into a single character string. For example, `uplo = 'U'`, `trans = 'T'`, and `diag = 'N'` for a triangular routine would be specified as `opts = 'UTN'`.

NOTE

Use only uppercase characters for the *opts* string.

n1, n2, n3, n4

Problem dimensions for the subroutine *name*; these may not all be required.

Output Parameters

value

If *value* ≥ 0 : the value of the parameter specified by *ispec*;

If *value* = $-k < 0$: the *k*-th argument had an illegal value.

Return Values

ilaenv returns *value*.

If *value* ≥ 0 : the value of the parameter specified by *ispec*;

If *value* = $-k < 0$: the *k*-th argument had an illegal value.

Application Notes

The following conventions have been used when calling *ilaenv* from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1, n2, n3, n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by *ilaenv* is checked for validity in the calling subroutine. For example, *ilaenv* is used to retrieve the optimal blocksize for *strtri* as follows:

```
nb := ilaenv( 1, 'strtri', strcat (uplo, diag), n, -1, -1, -1> );
if( nb <= 1 ) {
    nb := max( 1, n );
}
```

Below is an example of *ilaenv* usage in C language:

```
#include <stdio.h>
#include "mkl.h"

int main(void)
{
    int size = 1000;
    int ispec = 1;
    int dummy = -1;
    int blockSize1 = ilaenv(&ispec, "dsytrd", "U", &size, &dummy, &dummy, &dummy);
    int blockSize2 = ilaenv(&ispec, "dormtr", "LUN", &size, &size, &dummy, &dummy);
    printf("DSYTRD blocksize = %d\n", blockSize1);
    printf("DORMTR blocksize = %d\n", blockSize2);
    return 0;
}
```

See Also

[?hseqr](#)

?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

```
float LAPACKE_slamch (char cmach );  
double LAPACKE_dlamch (char cmach );
```

Include Files

- mkl.h

Description

The function ?lamch determines single precision and double precision machine parameters.

Input Parameters

cmach

Specifies the value to be returned by ?lamch:

= 'E' or 'e', *val* = *eps*
= 'S' or 's', *val* = *sfmin*
= 'B' or 'b', *val* = *base*
= 'P' or 'p', *val* = *eps*base*
= 'n' or 'n', *val* = *t*
= 'R' or 'r', *val* = *rnd*
= 'M' or 'm', *val* = *emin*
= 'U' or 'u', *val* = *rmin*
= 'L' or 'l', *val* = *emax*
= 'O' or 'o', *val* = *rmax*

where

eps = relative machine precision;

sfmin = safe minimum, such that $1/sfmin$ does not overflow;

base = base of the machine;

prec = $eps*base$;

t = number of (base) digits in the mantissa;

rnd = 1.0 when rounding occurs in addition, 0.0 otherwise;

emin = minimum exponent before (gradual) underflow;

rmin = $underflow_threshold - base**(emin-1)$;

emax = largest exponent before overflow;

rmax = $overflow_threshold - (base**emax)*(1-eps)$.

NOTE

You can use a character string for *cmach* instead of a single character in order to make your code more readable. The first character of the string determines the value to be returned. For example, 'Precision' is interpreted as 'p'.

Output Parameters

val Value returned by the function.

LAPACK Test Functions and Routines

This section describes LAPACK test functions and routines.

?lagge

Generates a general m -by- n matrix .

Syntax

```
lapack_int LAPACKE_slagge (int matrix_layout , lapack_int m , lapack_int n , lapack_int
kl , lapack_int ku , const float * d , float * a , lapack_int lda , lapack_int *
iseed );
```

```
lapack_int LAPACKE_dlagge (int matrix_layout , lapack_int m , lapack_int n , lapack_int
kl , lapack_int ku , const double * d , double * a , lapack_int lda , lapack_int *
iseed );
```

```
lapack_int LAPACKE_clagge (int matrix_layout , lapack_int m , lapack_int n , lapack_int
kl , lapack_int ku , const float * d , lapack_complex_float * a , lapack_int lda ,
lapack_int * iseed );
```

```
lapack_int LAPACKE_zlagge (int matrix_layout , lapack_int m , lapack_int n , lapack_int
kl , lapack_int ku , const double * d , lapack_complex_double * a , lapack_int lda ,
lapack_int * iseed );
```

Include Files

- mkl.h

Description

The routine generates a general m -by- n matrix A , by pre- and post- multiplying a real diagonal matrix D with random matrices U and V :

$$A := U * D * V,$$

where U and V are orthogonal for real flavors and unitary for complex flavors. The lower and upper bandwidths may then be reduced to kl and ku by additional orthogonal transformations.

Input Parameters

A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

m	The number of rows of the matrix A ($m \geq 0$).
n	The number of columns of the matrix A ($n \geq 0$).
kl	The number of nonzero subdiagonals within the band of A ($0 \leq kl \leq m-1$).
ku	The number of nonzero superdiagonals within the band of A ($0 \leq ku \leq n-1$).
d	The array d with the dimension of $(\min(m, n))$ contains the diagonal elements of the diagonal matrix D .
lda	The leading dimension of the array a ($lda \geq m$) for column major layout and ($lda \geq n$) for row major layout.
$iseed$	The array $iseed$ with the dimension of 4 contains the seed of the random number generator. The elements must be between 0 and 4095 and $iseed$ must be odd.

Output Parameters

a	The array a with size at least $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout contains the generated m -by- n matrix A .
$iseed$	The array $iseed$ contains the updated seed on exit.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info < 0$, the i -th parameter had an illegal value.

If $info = -1011$, memory allocation error occurred.

?laghe

Generates a complex Hermitian matrix .

Syntax

```
lapack_int LAPACKE_claghe (int matrix_layout , lapack_int n , lapack_int k , const
float * d , lapack_complex_float * a , lapack_int lda , lapack_int * iseed );

lapack_int LAPACKE_zlaghe (int matrix_layout , lapack_int n , lapack_int k , const
double * d , lapack_complex_double * a , lapack_int lda , lapack_int * iseed );
```

Include Files

- mkl.h

Description

The routine generates a complex Hermitian matrix A , by pre- and post- multiplying a real diagonal matrix D with random unitary matrix:

$$A := U^* D^* U^H$$

The semi-bandwidth may then be reduced to k by additional unitary transformations.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>n</code>	The order of the matrix A ($n \geq 0$).
<code>k</code>	The number of nonzero subdiagonals within the band of A ($0 \leq k \leq n-1$).
<code>d</code>	The array d with the dimension of (n) contains the diagonal elements of the diagonal matrix D .
<code>lda</code>	The leading dimension of the array a ($lda \geq n$).
<code>iseed</code>	The array <code>iseed</code> with the dimension of 4 contains the seed of the random number generator. The elements must be between 0 and 4095 and <code>iseed[3]</code> must be odd.

Output Parameters

<code>a</code>	The array a of size at least $\max(1, lda * n)$ contains the generated n -by- n Hermitian matrix D .
<code>iseed</code>	The array <code>iseed</code> contains the updated seed on exit.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info < 0`, the i -th parameter had an illegal value.

If `info = -1011`, memory allocation error occurred.

?lagsy

Generates a symmetric matrix by pre- and post-multiplying a real diagonal matrix with a random unitary matrix.

Syntax

```
lapack_int LAPACKE_slagsy (int matrix_layout , lapack_int n , lapack_int k , const
float * d , float * a , lapack_int lda , lapack_int * iseed );
```

```
lapack_int LAPACKE_dlagsy (int matrix_layout , lapack_int n , lapack_int k , const
double * d , double * a , lapack_int lda , lapack_int * iseed );
```

```
lapack_int LAPACKE_clagsy (int matrix_layout , lapack_int n , lapack_int k , const
float * d , lapack_complex_float * a , lapack_int lda , lapack_int * iseed );
```

```
lapack_int LAPACKE_zlagsy (int matrix_layout , lapack_int n , lapack_int k , const
double * d , lapack_complex_double * a , lapack_int lda , lapack_int * iseed );
```

Include Files

- `mkl.h`

Description

The `?lagsy` routine generates a symmetric matrix A by pre- and post- multiplying a real diagonal matrix D with a random matrix U :

$$A := U^* D^* U^T,$$

where U is orthogonal for real flavors and unitary for complex flavors. The semi-bandwidth may then be reduced to k by additional unitary transformations.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

n	The order of the matrix A ($n \geq 0$).
k	The number of nonzero subdiagonals within the band of A ($0 \leq k \leq n-1$).
d	The array d with the dimension of (n) contains the diagonal elements of the diagonal matrix D .
lda	The leading dimension of the array a ($lda \geq n$).
$iseed$	The array $iseed$ with the dimension of 4 contains the seed of the random number generator. The elements must be between 0 and 4095 and $iseed[3]$ must be odd.

Output Parameters

a	The array a of size $\max(1, lda * n)$ contains the generated symmetric n -by- n matrix D .
$iseed$	The array $iseed$ contains the updated seed on exit.

Return Values

This function returns a value *info*.

If $info = 0$, the execution is successful.

If $info < 0$, the i -th parameter had an illegal value.

If $info = -1011$, memory allocation error occurred.

?latms

Generates a general m -by- n matrix with specific singular values.

Syntax

```
lapack_int LAPACKE_slatms (int matrix_layout, lapack_int m, lapack_int n, char dist,
lapack_int * iseed, char sym, float * d, lapack_int mode, float cond, float dmax,
lapack_int kl, lapack_int ku, char pack, float * a, lapack_int lda);
```

```
lapack_int LAPACKE_dlatms (int matrix_layout, lapack_int m, lapack_int n, char dist,
lapack_int * iseed, char sym, double * d, lapack_int mode, double cond, double dmax,
lapack_int kl, lapack_int ku, char pack, double * a, lapack_int lda);
```

```
lapack_int LAPACKE_clatms (int matrix_layout, lapack_int m, lapack_int n, char dist,
lapack_int * iseed, char sym, float * d, lapack_int mode, float cond, float dmax,
lapack_int kl, lapack_int ku, char pack, lapack_complex_float * a, lapack_int lda);

lapack_int LAPACKE_zlatms (int matrix_layout, lapack_int m, lapack_int n, char dist,
lapack_int * iseed, char sym, double * d, lapack_int mode, double cond, double dmax,
lapack_int kl, lapack_int ku, char pack, lapack_complex_double * a, lapack_int lda);
```

Include Files

- mkl.h

Description

The `?latms` routine generates random matrices with specified singular values, or symmetric/Hermitian matrices with specified eigenvalues for testing LAPACK programs.

It applies this sequence of operations:

1. Set the diagonal to d , where d is input or computed according to $mode$, $cond$, $dmax$, and sym as described in Input Parameters.
2. Generate a matrix with the appropriate band structure, by one of two methods:

Method A

1. Generate a dense m -by- n matrix by multiplying d on the left and the right by random unitary matrices, then:
2. Reduce the bandwidth according to kl and ku , using Householder transformations.

Method B:

Convert the bandwidth-0 (i.e., diagonal) matrix to a bandwidth-1 matrix using Givens rotations, "chasing" out-of-band elements back, much as in QR; then convert the bandwidth-1 to a bandwidth-2 matrix, etc.

Note that for reasonably small bandwidths (relative to m and n) this requires less storage, as a dense matrix is not generated. Also, for symmetric or Hermitian matrices, only one triangle is generated.

Method A is chosen if the bandwidth is a large fraction of the order of the matrix, and lda is at least m (so a dense matrix can be stored.) Method B is chosen if the bandwidth is small (less than $(1/2)*n$ for symmetric or Hermitian or less than $.3*n+m$ for nonsymmetric), or lda is less than m and not less than the bandwidth.

Pack the matrix if desired, using one of the methods specified by the `pack` parameter.

If Method B is chosen and band format is specified, then the matrix is generated in the band format and no repacking is necessary.

Input Parameters

A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See [C Interface Conventions](#) for the C interface principal conventions and type definitions.

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>m</code>	The number of rows of the matrix A ($m \geq 0$).
<code>n</code>	The number of columns of the matrix A ($n \geq 0$).

<i>dist</i>	<p>Specifies the type of distribution to be used to generate the random singular values or eigenvalues:</p> <ul style="list-style-type: none"> • 'U': uniform distribution (0, 1) • 'S': symmetric uniform distribution (-1, 1) • 'N': normal distribution (0, 1)
<i>iseed</i>	<p>Array with size 4.</p> <p>Specifies the seed of the random number generator. Values should lie between 0 and 4095 inclusive, and <i>iseed</i>[3] should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers. The values of the array are modified, and can be used in the next call to ?latms to continue the same random number sequence.</p>
<i>sym</i>	<p>If <i>sym</i>='S' or 'H', the generated matrix is symmetric or Hermitian, with eigenvalues specified by <i>d</i>, <i>cond</i>, <i>mode</i>, and <i>dmax</i>; they can be positive, negative, or zero.</p> <p>If <i>sym</i>='P', the generated matrix is symmetric or Hermitian, with eigenvalues (which are singular, non-negative values) specified by <i>d</i>, <i>cond</i>, <i>mode</i>, and <i>dmax</i>.</p> <p>If <i>sym</i>='N', the generated matrix is nonsymmetric, with singular, non-negative values specified by <i>d</i>, <i>cond</i>, <i>mode</i>, and <i>dmax</i>.</p>
<i>d</i>	<p>Array, size (MIN(<i>m</i>, <i>n</i>))</p> <p>This array is used to specify the singular values or eigenvalues of <i>A</i> (see the description of <i>sym</i>). If <i>mode</i>=0, then <i>d</i> is assumed to contain the eigenvalues or singular values, otherwise elements of <i>d</i> are computed according to <i>mode</i>, <i>cond</i>, and <i>dmax</i>.</p>
<i>mode</i>	<p>Describes how the singular/eigenvalues are specified.</p> <ul style="list-style-type: none"> • <i>mode</i> = 0: use <i>d</i> as input • <i>mode</i> = 1: set $d[0] = 1$ and $d[1:n - 1] = 1.0/cond$ • <i>mode</i> = 2: set $d[0:n - 2] = 1$ and $d[n - 1] = 1.0/cond$ • <i>mode</i> = 3: set $d[i] = cond^{-i/(n - 1)}$ • <i>mode</i> = 4: set $d[i] = 1 - i/(n - 1) * (1 - 1/cond)$ • <i>mode</i> = 5: set elements of <i>d</i> to random numbers in the range (1/<i>cond</i>, 1) such that their logarithms are uniformly distributed. • <i>mode</i> = 6: set elements of <i>d</i> to random numbers from same distribution as the rest of the matrix. <p><i>mode</i> < 0 has the same meaning as ABS(<i>mode</i>), except that the order of the elements of <i>d</i> is reversed. Thus, if <i>mode</i> is positive, <i>d</i> has entries ranging from 1 to 1/<i>cond</i>, if negative, from 1/<i>cond</i> to 1.</p> <p>If <i>sym</i>='S' or 'H', and <i>mode</i> is not 0, 6, nor -6, then the elements of <i>d</i> are also given a random sign (multiplied by +1 or -1).</p>
<i>cond</i>	Used in setting <i>d</i> as described for the <i>mode</i> parameter. If used, $cond \geq 1$.
<i>dmax</i>	If <i>mode</i> is not -6, 0 nor 6, the contents of <i>d</i> , as computed according to <i>mode</i> and <i>cond</i> , are scaled by $dmax / \max(\text{abs}(d[i-1]))$; thus, the maximum absolute eigenvalue or singular value (the norm) is $\text{abs}(dmax)$.

NOTE

d_{max} need not be positive: if d_{max} is negative (or zero), d will be scaled by a negative number (or zero).

 kl

Specifies the lower bandwidth of the matrix. For example, $kl=0$ implies upper triangular, $kl=1$ implies upper Hessenberg, and kl being at least $m - 1$ means that the matrix has full lower bandwidth. kl must equal ku if the matrix is symmetric or Hermitian.

 ku

Specifies the upper bandwidth of the matrix. For example, $ku=0$ implies lower triangular, $ku=1$ implies lower Hessenberg, and ku being at least $n - 1$ means that the matrix has full upper bandwidth. kl must equal ku if the matrix is symmetric or Hermitian.

 $pack$

Specifies packing of matrix:

- 'N': no packing
- 'U': zero out all subdiagonal entries (if symmetric or Hermitian)
- 'L': zero out all superdiagonal entries (if symmetric or Hermitian)
- 'B': store the lower triangle in band storage scheme (only if matrix symmetric, Hermitian, or lower triangular)
- 'Q': store the upper triangle in band storage scheme (only if matrix symmetric, Hermitian, or upper triangular)
- 'Z': store the entire matrix in band storage scheme (pivoting can be provided for by using this option to store A in the trailing rows of the allocated storage)

Using these options, the various LAPACK packed and banded storage schemes can be obtained:

	'Z'	'B'	'Q'	'C'	'R'
GB: general band	x				
PB: symmetric positive definite band		x	x		
SB: symmetric band		x	x		
HB: Hermitian band		x	x		
TB: triangular band		x	x		
PP: symmetric positive definite packed				x	x
SP: symmetric packed				x	x
HP: Hermitian packed				x	x
TP: triangular packed				x	x

If two calls to `?latms` differ only in the $pack$ parameter, they generate mathematically equivalent matrices.

 lda

lda specifies the first dimension of a as declared in the calling program.

If *pack*='N', 'U', 'L', 'C', or 'R', then *lda* must be at least *m* for column major or at least *n* for row major.

If *pack*='B' or 'Q', then *lda* must be at least $\text{MIN}(kl, m - 1)$ (which is equal to $\text{MIN}(ku, n - 1)$).

If *pack*='Z', *lda* must be large enough to hold the packed array: $\text{MIN}(ku, n - 1) + \text{MIN}(kl, m - 1) + 1$.

Output Parameters

iseed

The array *iseed* contains the updated seed.

d

The array *d* contains the updated seed.

NOTE

The array *d* is not modified if *mode* = 0.

a

Array of size *lda* by *n*.

The array *a* contains the generated *m*-by-*n* matrix *A*.

a is first generated in full (unpacked) form, and then packed, if so specified by *pack*. Thus, the first *m* elements of the first *n* columns are always modified. If *pack* specifies a packed or banded storage scheme, all *lda* elements of the first *n* columns are modified; the elements of the array which do not correspond to elements of the generated matrix are set to zero.

Return Values

This function returns a value *info*.

If *info* = 0, the execution is successful.

If *info* < 0, the *i*-th parameter had an illegal value.

If *info* = -1011, memory allocation error occurred.

If *info* = 2, cannot scale to *dmax* (maximum singular value is 0).

If *info* = 3, error return from [lagge](#), [?laghe](#), or [lagsy](#).

Additional LAPACK Routines (Included for Compatibility with Netlib LAPACK)

```
LAPACK_DECL lapack_int LAPACKE_chesv_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , lapack_complex_float * a , lapack_int lda ,
lapack_complex_float * tb , lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 ,
lapack_complex_float * b , lapack_int ldb );
```

```
LAPACK_DECL lapack_int LAPACKE_dsysv_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , double * a , lapack_int lda , double * tb , lapack_int
ltb , lapack_int * ipiv , lapack_int * ipiv2 , double * b , lapack_int ldb );
```

```
LAPACK_DECL lapack_int LAPACKE_ssysv_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , float * a , lapack_int lda , float * tb , lapack_int
ltb , lapack_int * ipiv , lapack_int * ipiv2 , float * b , lapack_int ldb );
```



```

LAPACK_DECL lapack_int LAPACKE_zhesv_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , lapack_complex_double * a , lapack_int lda ,
lapack_complex_double * tb , lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 ,
lapack_complex_double * b , lapack_int ldb );

LAPACK_DECL lapack_int LAPACKE_chetrf_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_complex_float * a , lapack_int lda , lapack_complex_float * tb ,
lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 );

LAPACK_DECL lapack_int LAPACKE_dsytrf_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , double * a , lapack_int lda , double * tb , lapack_int ltb , lapack_int
* ipiv , lapack_int * ipiv2 );

LAPACK_DECL lapack_int LAPACKE_ssytrf_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , float * a , lapack_int lda , float * tb , lapack_int ltb , lapack_int *
ipiv , lapack_int * ipiv2 );

LAPACK_DECL lapack_int LAPACKE_zhetrf_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_complex_double * a , lapack_int lda , lapack_complex_double *
tb , lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 );

LAPACK_DECL lapack_int LAPACKE_chetrs_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , lapack_complex_float * a , lapack_int lda ,
lapack_complex_float * tb , lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 ,
lapack_complex_float * b , lapack_int ldb );

LAPACK_DECL lapack_int LAPACKE_dsytrs_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , double * a , lapack_int lda , double * tb , lapack_int
ltb , lapack_int * ipiv , lapack_int * ipiv2 , double * b , lapack_int ldb );

LAPACK_DECL lapack_int LAPACKE_ssytrs_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , float * a , lapack_int lda , float * tb , lapack_int
ltb , lapack_int * ipiv , lapack_int * ipiv2 , float * b , lapack_int ldb );

LAPACK_DECL lapack_int LAPACKE_zhetrs_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , lapack_complex_double * a , lapack_int lda ,
lapack_complex_double * tb , lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 ,
lapack_complex_double * b , lapack_int ldb );

call csysv_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , b , ldb ,
info);

LAPACK_DECL lapack_int LAPACKE_csysv_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , lapack_complex_float * a , lapack_int lda ,
lapack_complex_float * tb , lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 ,
lapack_complex_float * b , lapack_int ldb );

call zsysv_aa_2stage (uplo , n , nrhs , a , lda , tb , ltb , ipiv , ipiv2 , b , ldb ,
info);

LAPACK_DECL lapack_int LAPACKE_zsysv_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , lapack_complex_double * a , lapack_int lda ,
lapack_complex_double * tb , lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 ,
lapack_complex_double * b , lapack_int ldb );

LAPACK_DECL lapack_int LAPACKE_csytrf_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_complex_float * a , lapack_int lda , lapack_complex_float * tb ,
lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 );

LAPACK_DECL lapack_int LAPACKE_zsytrf_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_complex_double * a , lapack_int lda , lapack_complex_double *
tb , lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 );

```

```

LAPACK_DECL lapack_int LAPACKE_csytrs_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , lapack_complex_float * a , lapack_int lda ,
lapack_complex_float * tb , lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 ,
lapack_complex_float * b , lapack_int ldb );

LAPACK_DECL lapack_int LAPACKE_zsytrs_aa_2stage (int matrix_layout , char uplo ,
lapack_int n , lapack_int nrhs , lapack_complex_double * a , lapack_int lda ,
lapack_complex_double * tb , lapack_int ltb , lapack_int * ipiv , lapack_int * ipiv2 ,
lapack_complex_double * b , lapack_int ldb );

LAPACK_DECL lapack_int LAPACKE_ssyev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, float * a, lapack_int lda, float * w);

LAPACK_DECL lapack_int LAPACKE_dsyevev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, double * a, lapack_int lda, double * w);

LAPACK_DECL lapack_int LAPACKE_ssyevd_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, float * a, lapack_int lda, float * w);

LAPACK_DECL lapack_int LAPACKE_dsyevd_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, double * a, lapack_int lda, double * w);

LAPACK_DECL lapack_int LAPACKE_ssyevr_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, float * a, lapack_int lda, float vl, float vu, lapack_int il,
lapack_int iu, float abstol, lapack_int * m, float * w, float * z, lapack_int ldz,
lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_dsyevr_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, double * a, lapack_int lda, double vl, double vu, lapack_int
il, lapack_int iu, double abstol, lapack_int * m, double * w, double * z, lapack_int
ldz, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_ssyevx_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, float * a, lapack_int lda, float vl, float vu, lapack_int il,
lapack_int iu, float abstol, lapack_int * m, float * w, float * z, lapack_int ldz,
lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_dsyevx_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, double * a, lapack_int lda, double vl, double vu, lapack_int
il, lapack_int iu, double abstol, lapack_int * m, double * w, double * z, lapack_int
ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_ssygv_2stage (int matrix_layout, lapack_int itype, char
jobz, char uplo, lapack_int n, float * a, lapack_int lda, float * b, lapack_int ldb,
float * w);

LAPACK_DECL lapack_int LAPACKE_dsygv_2stage (int matrix_layout, lapack_int itype, char
jobz, char uplo, lapack_int n, double * a, lapack_int lda, double * b, lapack_int ldb,
double * w);

LAPACK_DECL lapack_int LAPACKE_cheev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_complex_float * a, lapack_int lda, float * w);

LAPACK_DECL lapack_int LAPACKE_zheev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_complex_double * a, lapack_int lda, double * w);

LAPACK_DECL lapack_int LAPACKE_cheevd_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_complex_float * a, lapack_int lda, float * w);

LAPACK_DECL lapack_int LAPACKE_zheevd_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_complex_double * a, lapack_int lda, double * w);

LAPACK_DECL lapack_int LAPACKE_cheevr_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda, float vl, float vu,
lapack_int il, lapack_int iu, float abstol, lapack_int * m, float * w,
lapack_complex_float * z, lapack_int ldz, lapack_int * isuppz);

```

```

LAPACK_DECL lapack_int LAPACKE_zheevr_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, double vl, double
vu, lapack_int il, lapack_int iu, double abstol, lapack_int * m, double * w,
lapack_complex_double * z, lapack_int ldz, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_cheevx_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda, float vl, float vu,
lapack_int il, lapack_int iu, float abstol, lapack_int * m, float * w,
lapack_complex_float * z, lapack_int ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_zheevx_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, double vl, double
vu, lapack_int il, lapack_int iu, double abstol, lapack_int * m, double * w,
lapack_complex_double * z, lapack_int ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_chegv_2stage (int matrix_layout, lapack_int itype, char
jobz, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda,
lapack_complex_float * b, lapack_int ldb, float * w);

LAPACK_DECL lapack_int LAPACKE_zhegv_2stage (int matrix_layout, lapack_int itype, char
jobz, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda,
lapack_complex_double * b, lapack_int ldb, double * w);

LAPACK_DECL lapack_int LAPACKE_ssbv_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, float * ab, lapack_int ldab, float * w, float * z,
lapack_int ldz);

LAPACK_DECL lapack_int LAPACKE_dsbev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, double * ab, lapack_int ldab, double * w, double * z,
lapack_int ldz);

LAPACK_DECL lapack_int LAPACKE_ssbv_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, float * ab, lapack_int ldab, float * w, float * z,
lapack_int ldz);

LAPACK_DECL lapack_int LAPACKE_dsbev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, double * ab, lapack_int ldab, double * w, double * z,
lapack_int ldz);

LAPACK_DECL lapack_int LAPACKE_ssbvx_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, lapack_int kd, float * ab, lapack_int ldab, float * q,
lapack_int ldq, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int * m, float * w, float * z, lapack_int ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_dsbev_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, lapack_int kd, double * ab, lapack_int ldab, double * q,
lapack_int ldq, double vl, double vu, lapack_int il, lapack_int iu, double abstol,
lapack_int * m, double * w, double * z, lapack_int ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_chbev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, lapack_complex_float * ab, lapack_int ldab, float * w,
lapack_complex_float * z, lapack_int ldz);

LAPACK_DECL lapack_int LAPACKE_zhbev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, lapack_complex_double * ab, lapack_int ldab, double * w,
lapack_complex_double * z, lapack_int ldz);

LAPACK_DECL lapack_int LAPACKE_chbev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, lapack_complex_float * ab, lapack_int ldab, float * w,
lapack_complex_float * z, lapack_int ldz);

LAPACK_DECL lapack_int LAPACKE_zhbev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, lapack_complex_double * ab, lapack_int ldab, double * w,
lapack_complex_double * z, lapack_int ldz);

```

```
LAPACK_DECL lapack_int LAPACKE_chbev_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, lapack_int kd, lapack_complex_float * ab, lapack_int ldab,
lapack_complex_float * q, lapack_int ldq, float vl, float vu, lapack_int il, lapack_int
iu, float abstol, lapack_int * m, float * w, lapack_complex_float * z, lapack_int ldz,
lapack_int * ifail);
```

```
LAPACK_DECL lapack_int LAPACKE_zhbev_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, lapack_int kd, lapack_complex_double * ab, lapack_int ldab,
lapack_complex_double * q, lapack_int ldq, double vl, double vu, lapack_int il,
lapack_int iu, double abstol, lapack_int * m, double * w, lapack_complex_double * z,
lapack_int ldz, lapack_int * ifail);
```

For descriptions of these functions, please see <https://www.netlib.org/lapack/explore-html/files.html>.

ScaLAPACK Routines

Intel® oneAPI Math Kernel Library implements routines from the ScaLAPACK package for distributed-memory architectures. Routines are supported for both real and complex dense and band matrices to perform the tasks of solving systems of linear equations, solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

Intel® oneAPI Math Kernel Library (oneMKL) ScaLAPACK routines are written in FORTRAN 77 with exception of a few utility routines written in C to exploit the IEEE arithmetic. All routines are available in all precision types: single precision, double precision, complex, and double complex precision. See `themkl_scalapack.h` header file for C declarations of ScaLAPACK routines.

NOTE

ScaLAPACK routines are provided only for Intel® 64 or Intel® Many Integrated Core architectures.

See descriptions of ScaLAPACK [computational routines](#) that perform distinct computational tasks, as well as [driver routines](#) for solving standard types of problems in one call. Additionally, Intel® oneAPI Math Kernel Library implements ScaLAPACK [Auxiliary Routines](#), [Utility Functions and Routines](#), and [Matrix Redistribution/Copy Routines](#). The library includes routines for both real and complex data.

The `<install_directory>/examples/scalapackf` directory contains sample code demonstrating the use of ScaLAPACK routines.

Generally, ScaLAPACK runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of BLAS optimized for the target architecture. Intel® oneAPI Math Kernel Library (oneMKL) version of ScaLAPACK is optimized for Intel® processors. For the detailed system and environment requirements, see *Intel® oneAPI Math Kernel Library (oneMKL) Release Notes* and *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

For full reference on ScaLAPACK routines and related information, see [\[SLUG\]](#).

Product and Performance Information

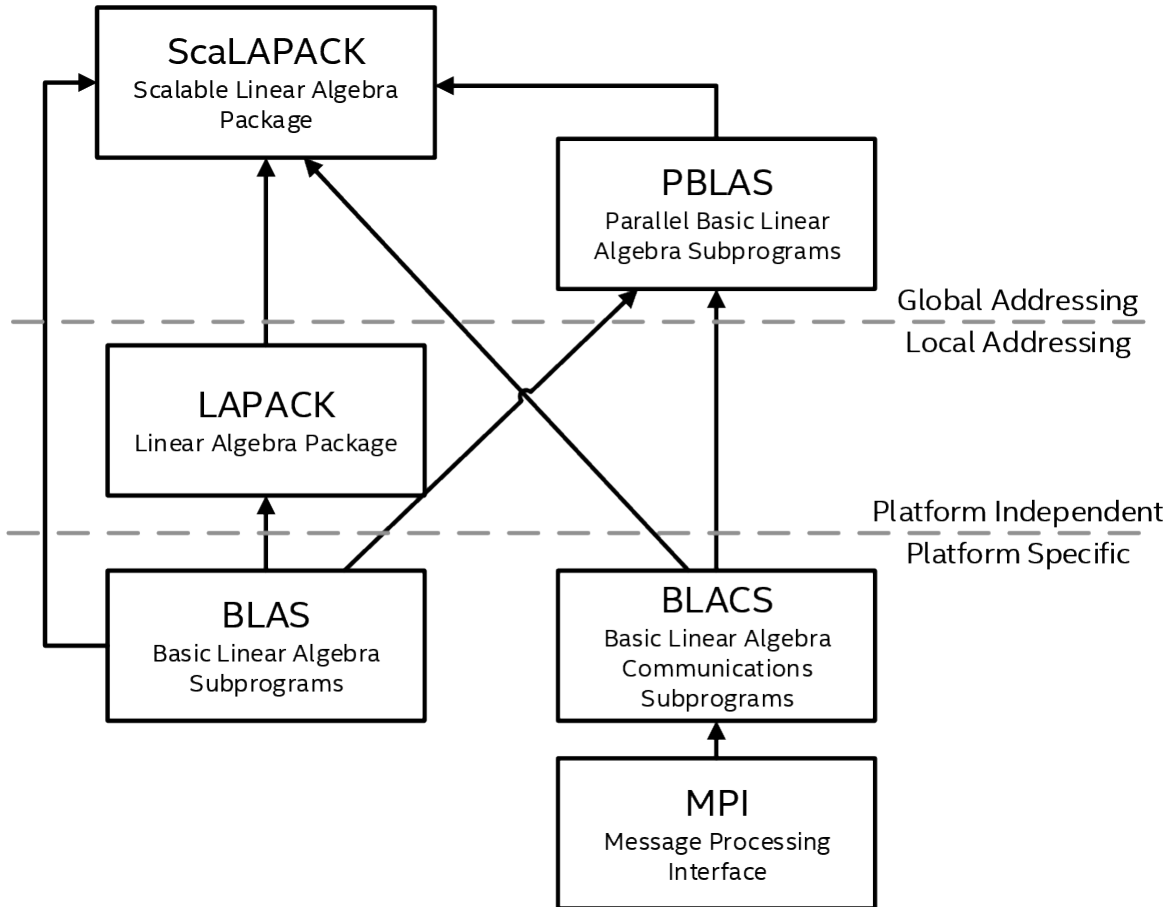
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

Overview of ScaLAPACK Routines

The model of the computing environment for ScaLAPACK is represented as a one-dimensional array of processes (for operations on band or tridiagonal matrices) or also a two-dimensional process grid (for operations on dense matrices). To use ScaLAPACK, all global matrices or vectors should be distributed on this array or grid prior to calling the ScaLAPACK routines.

ScaLAPACK is closely tied to other components, including BLAS, BLACS, LAPACK, and PBLAS.



ScaLAPACK Array Descriptors

ScaLAPACK uses two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, and also allows use of BLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global matrix and its corresponding process and memory location is contained in the array called the *array descriptor* associated with each global matrix. The size of the array descriptor is denoted as *dlen_*.

Let A be a two-dimensional block cyclicly distributed matrix with the array descriptor array *desca*. The meaning of each array descriptor element depends on the type of the matrix A . The tables "Array descriptor for dense matrices" and "Array descriptor for narrow-band and tridiagonal matrices" describe the meaning of each element for the different types of matrices.

Array descriptor for dense matrices (*dlen_=9*)

Element Name	Stored in	Description	Element Index Number
<i>dtype_a</i>	<i>desca[dtype_]</i>	Descriptor type (=1 for dense matrices).	0
<i>ctxt_a</i>	<i>desca[ctxt_]</i>	BLACS context handle for the process grid.	1
<i>m_a</i>	<i>desca[m_]</i>	Number of rows in the global matrix A .	2
<i>n_a</i>	<i>desca[n_]</i>	Number of columns in the global matrix A .	3
<i>mb_a</i>	<i>desca[mb_]</i>	Row blocking factor.	4

Element Name	Stored in	Description	Element Index Number
<i>nb_a</i>	<i>desca[nb_]</i>	Column blocking factor.	5
<i>rsrc_a</i>	<i>desca[rsrc_]</i>	Process row over which the first row of the global matrix <i>A</i> is distributed.	6
<i>csrc_a</i>	<i>desca[csrc_]</i>	Process column over which the first column of the global matrix <i>A</i> is distributed.	7
<i>lld_a</i>	<i>desca[lld_]</i>	Leading dimension of the local matrix <i>A</i> .	8

Array descriptor for narrow-band and tridiagonal matrices (*dlen_*=7)

Element Name	Stored in	Description	Element Index Number
<i>dtype_a</i>	<i>desca[dtype_]</i>	Descriptor type <ul style="list-style-type: none"> <i>dtype_a</i>=501: 1-by-<i>P</i> grid, <i>dtype_a</i>=502: <i>P</i>-by-1 grid. 	0
<i>ctxt_a</i>	<i>desca[ctxt_]</i>	BLACS context handle indicating the BLACS process grid over which the global matrix <i>A</i> is distributed. The context itself is global, but the handle (the integer value) can vary.	1
<i>n_a</i>	<i>desca[n_]</i>	The size of the matrix dimension being distributed.	2
<i>nb_a</i>	<i>desca[nb_]</i>	The blocking factor used to distribute the distributed dimension of the matrix <i>A</i> .	3
<i>src_a</i>	<i>desca[src_]</i>	The process row or column over which the first row or column of the matrix <i>A</i> is distributed.	4
<i>lld_a</i>	<i>desca[lld_]</i>	The leading dimension of the local matrix storing the local blocks of the distributed matrix <i>A</i> . The minimum value of <i>lld_a</i> depends on <i>dtype_a</i> . <ul style="list-style-type: none"> <i>dtype_a</i>=501: $lld_a \geq \max(\text{size of undistributed dimension}, 1)$, <i>dtype_a</i>=502: $lld_a \geq \max(nb_a, 1)$. 	5
Not applicable		Reserved for future use.	6

Similar notations are used for different matrices. For example: *lld_b* is the leading dimension of the local matrix storing the local blocks of the distributed matrix *B* and *dtype_z* is the type of the global matrix *Z*.

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by *LOC_r*() and *LOC_c*(), respectively. To compute these numbers, you can use the ScaLAPACK tool routine *numroc*.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix sub(*A*) of the global matrix *A* defined by the following 6 values (for dense matrices):

<i>m</i>	The number of rows of sub(<i>A</i>)
<i>n</i>	The number of columns of sub(<i>A</i>)
<i>a</i>	A pointer to the local matrix containing the entire global matrix <i>A</i>
<i>ia</i>	The row index of sub(<i>A</i>) in the global matrix <i>A</i>
<i>ja</i>	The column index of sub(<i>A</i>) in the global matrix <i>A</i>

desca The array descriptor for the global matrix A

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

Naming Conventions for ScaLAPACK Routines

For each routine introduced in this chapter, you can use the ScaLAPACK name. The naming convention for ScaLAPACK routines is similar to that used for LAPACK routines. A general rule is that each routine name in ScaLAPACK, which has an LAPACK equivalent, is simply the LAPACK name prefixed by initial letter *p*.

ScaLAPACK names have the structure *p?yyzzz* or *p?yyzz*, which is described below.

The initial letter *p* is a distinctive prefix of ScaLAPACK routines and is present in each such routine.

The second symbol *?* indicates the data type:

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

The second and third letters *yy* indicate the matrix type as:

ge	general
gb	general band
gg	a pair of general matrices (for a generalized problem)
dt	general tridiagonal (diagonally dominant-like)
db	general band (diagonally dominant-like)
po	symmetric or Hermitian positive-definite
pb	symmetric or Hermitian positive-definite band
pt	symmetric or Hermitian positive-definite tridiagonal
sy	symmetric
st	symmetric tridiagonal (real)
he	Hermitian
or	orthogonal
tr	triangular (or quasi-triangular)
tz	trapezoidal
un	unitary

For computational routines, the last three letters **zzz** indicate the computation performed and have the same meaning as for LAPACK routines.

For driver routines, the last two letters **zz** or three letters **zzz** have the following meaning:

<code>sv</code>	a <i>simple</i> driver for solving a linear system
<code>svx</code>	an <i>expert</i> driver for solving a linear system
<code>ls</code>	a driver for solving a linear least squares problem
<code>ev</code>	a simple driver for solving a symmetric eigenvalue problem
<code>evd</code>	a simple driver for solving an eigenvalue problem using a divide and conquer algorithm
<code>evx</code>	an expert driver for solving a symmetric eigenvalue problem
<code>svd</code>	a driver for computing a singular value decomposition
<code>gvx</code>	an expert driver for solving a generalized symmetric definite eigenvalue problem

Simple driver here means that the driver just solves the general problem, whereas an *expert* driver is more versatile and can also optionally perform some related computations (such, for example, as refining the solution and computing error bounds after the linear system is solved).

ScaLAPACK Computational Routines

In the sections that follow, the descriptions of ScaLAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

- [Solving Systems of Linear Equations](#)
- [Orthogonal Factorizations and LLS Problems](#)
- [Symmetric Eigenproblems](#)
- [Nonsymmetric Eigenproblems](#)
- [Singular Value Decomposition](#)
- [Generalized Symmetric-Definite Eigenproblems](#)

See also the respective [driver routines](#).

Systems of Linear Equations: ScaLAPACK Computational Routines

ScaLAPACK supports routines for the systems of equations with the following types of matrices:

- general
- general banded
- general diagonally dominant-like banded (including general tridiagonal)
- symmetric or Hermitian positive-definite
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian positive-definite tridiagonal

A *diagonally dominant-like* matrix is defined as a matrix for which it is known in advance that pivoting is not required in the *LU* factorization of this matrix.

For the above matrix types, the library includes routines for performing the following computations: *factoring* the matrix; *equilibrating* the matrix; *solving* a system of linear equations; *estimating the condition number* of a matrix; *refining* the solution of linear equations and computing its error bounds; *inverting* the matrix. Note that for some of the listed matrix types only part of the computational routines are provided (for example, routines that refine the solution are not provided for band or tridiagonal matrices). See [Table "Computational Routines for Systems of Linear Equations"](#) for full list of available routines.

To solve a particular problem, you can either call two or more computational routines or call a corresponding [driver routine](#) that combines several tasks in one call. Thus, to solve a system of linear equations with a general matrix, you can first call `p?getrf` (*LU* factorization) and then `p?getrs` (computing the solution). Then, you might wish to call `p?gerfs` to refine the solution and get the error bounds. Alternatively, you can just use the driver routine `p?gesvx` which performs all these tasks in one call.

Table “Computational Routines for Systems of Linear Equations” lists the ScaLAPACK computational routines for factorizing, equilibrating, and inverting matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error.

Computational Routines for Systems of Linear Equations

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general (partial pivoting)	p?getrf	p?geequ	p?getrs	p?gecon	p?gerfs	p?getri
general band (partial pivoting)	p?gbtrf		p?gbtrs			
general band (no pivoting)	p?dbtrf		p?dbtrs			
general tridiagonal (no pivoting)	p?dttrf		p?dttrs			
symmetric/Hermitian positive-definite	p?potrf	p?poequ	p?potrs	p?pocon	p?porfs	p?potri
symmetric/Hermitian positive-definite, band	p?pbtrf		p?pbtrs			
symmetric/Hermitian positive-definite, tridiagonal	p?pttrf		p?pttrs			
triangular			p?trtrs	p?trcon	p?trrfs	p?trtri

In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex).

Matrix Factorization: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization of general matrices
- LU factorization of diagonally dominant-like matrices
- Cholesky factorization of real symmetric or complex Hermitian positive-definite matrices

You can compute the factorizations using full and band storage of matrices.

[p?getrf](#)

Computes the LU factorization of a general m-by-n distributed matrix.

Syntax

```
void psgetrf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
void pdgetrf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
void pcgetrf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
void pzgetrf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?getrf` function forms the LU factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia} + m - 1, \text{ja}:\text{ja} + n - 1)$ as

$$A = P * L * U$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). L and U are stored in $\text{sub}(A)$.

The function uses partial pivoting, with row interchanges.

NOTE

This function supports the Progress Routine feature. See [mkl_progress](#) for details.

Input Parameters

m	(global) The number of rows in the distributed matrix $\text{sub}(A)$; $m \geq 0$.
n	(global) The number of columns in the distributed matrix $\text{sub}(A)$; $n \geq 0$.
a	(local) Pointer into the local memory to an array of local size $ld_a * LOCc(\text{ja} + n - 1)$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
desca	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

a	Overwritten by local pieces of the factors L and U from the factorization $A = P * L * U$. The unit diagonal elements of L are not stored.
ipiv	(local) Array of size $LOCr(m_a) + mb_a$. Contains the pivoting information: local row i was interchanged with global row $\text{ipiv}[i - 1]$. This array is tied to the distributed matrix A .
info	(global) If $\text{info} = 0$, the execution is successful. $\text{info} < 0$: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then $\text{info} = -(i * 100 + j)$; if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$. If $\text{info} = i > 0$, $u_{\text{ia}+i, \text{ja}+j-1}$ is 0. The factorization has been completed, but the factor U is exactly singular. Division by zero will occur if you use the factor U for solving a system of linear equations.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gbtrf

Computes the LU factorization of a general n -by- n banded distributed matrix.

Syntax

```
void psgbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , float *a , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , float *af , MKL_INT *laf , float *work , MKL_INT
*lwork , MKL_INT *info );

void pdgbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , double *a , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , double *af , MKL_INT *laf , double *work , MKL_INT
*lwork , MKL_INT *info );

void pcgbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_Complex8 *a , MKL_INT
*ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8
*work , MKL_INT *lwork , MKL_INT *info );

void pzgbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_Complex16 *a , MKL_INT
*ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16
*work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?gbtrf` function computes the LU factorization of a general n -by- n real/complex banded distributed matrix $A(1:n, ja:ja+n-1)$ using partial pivoting with row interchanges.

The resulting factorization is not the same factorization as returned from the LAPACK function `?gbtrf`. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form

$$A(1:n, ja:ja+n-1) = P * L * U * Q$$

where P and Q are permutation matrices, and L and U are banded lower and upper triangular matrices, respectively. The matrix Q represents reordering of columns for the sake of parallelism, while P represents reordering of rows for numerical stability using classic partial pivoting.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

n	(global) The number of rows and columns in the distributed submatrix $A(1:n, ja:ja+n-1)$; $n \geq 0$.
bwl	(global) The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
bwu	(global) The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).
a	(local) Pointer into the local memory to an array of local size $lld_a * LOCc(ja+n-1)$ where

$lld_a \geq 2*bwl + 2*bwu + 1$.

Contains the local pieces of the n -by- n distributed banded matrix $A(1:n, ja:ja+n-1)$ to be factored.

ja (global) The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).

desca (global and local) array of size *dlen_*. The array descriptor for the distributed matrix A .

If $dtype_a = 501$, then $dlen_ \geq 7$;

else if $dtype_a = 1$, then $dlen_ \geq 9$.

laf (local) The size of the array *af*.

Must be $laf \geq (nb_a + bwu) * (bwl + bwu) + 6 * (bwl + bwu) * (bwl + 2 * bwu)$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*[0].

work (local) Same type as *a*. Workspace array of size *lwork*.

lwork (local or global) The size of the *work* array ($lwork \geq 1$). If *lwork* is too small, the minimal acceptable size will be returned in *work*[0] and an error code is returned.

Output Parameters

a On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

ipiv (local) array.

The size of *ipiv* must be $\geq nb_a$.

Contains pivot indices for local factorizations. Note that you *should not alter* the contents of this array between factorization and solve.

af (local)

Array of size *laf*.

Auxiliary fill-in space. The fill-in space is created in a call to the factorization function `p?gbtrf` and is stored in *af*.

Note that if a linear system is to be solved using `p?gbtrs` after the factorization function, *af* must not be altered after the factorization.

work[0] On exit, *work*[0] contains the minimum value of *lwork* required.

info (global)

If *info*=0, the execution is successful.

$info < 0$:

If the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then $info = -(i*100+j)$; if the *i*-th argument is a scalar and had an illegal value, then $info = -i$.

info > 0:

If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not nonsingular, and the factorization was not completed.

If *info* = *k* > NPROCS, the submatrix stored on processor *info* - NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dbtrf

Computes the LU factorization of a n -by- n diagonally dominant-like banded distributed matrix.

Syntax

```
void psdbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , float *a , MKL_INT *ja ,
MKL_INT *desca , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT
*info );
```

```
void pddbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , double *a , MKL_INT *ja ,
MKL_INT *desca , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT
*info );
```

```
void pcdbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_Complex8 *a , MKL_INT
*ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
```

```
void pzdbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_Complex16 *a , MKL_INT
*ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?dbtrf` function computes the LU factorization of a n -by- n real/complex diagonally dominant-like banded distributed matrix $A(1:n, ja:ja+n-1)$ without pivoting.

NOTE

A matrix is called *diagonally dominant-like* if pivoting is not required for LU to be numerically stable.

Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>n</i>	(global) The number of rows and columns in the distributed submatrix $A(1:n, ja:ja+n-1)$; $n \geq 0$.
<i>bwl</i>	(global) The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
<i>bwu</i>	(global) The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).
<i>a</i>	(local) Pointer into the local memory to an array of local size $ld_a * LOCC(ja+n-1)$. Contains the local pieces of the n -by- n distributed banded matrix $A(1:n, ja:ja+n-1)$ to be factored.
<i>ja</i>	(global) The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A . If <i>dtype_a</i> = 501, then <i>dlen_</i> ≥ 7 ; else if <i>dtype_a</i> = 1, then <i>dlen_</i> ≥ 9 .
<i>laf</i>	(local) The size of the array <i>af</i> . Must be $laf \geq NB * (bwl + bwu) + 6 * (\max(bwl, bwu))^2$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> [0].
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the <i>work</i> array, must be $lwork \geq (\max(bwl, bwu))^2$. If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i> [0] and an error code is returned.

Output Parameters

<i>a</i>	On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
<i>af</i>	(local) Array of size <i>laf</i> . Auxiliary fill-in space. The fill-in space is created in a call to the factorization function <code>p?dbtrf</code> and is stored in <i>af</i> . Note that if a linear system is to be solved using <code>p?dbtrs</code> after the factorization function, <i>af</i> must not be altered after the factorization.
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.

info (global)

If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:

If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not diagonally dominant-like, and the factorization was not completed.

If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dttrf

Computes the LU factorization of a diagonally dominant-like tridiagonal distributed matrix.

Syntax

```
void psdttrf (MKL_INT *n , float *dl , float *d , float *du , MKL_INT *ja , MKL_INT
*desca , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pddttrf (MKL_INT *n , double *dl , double *d , double *du , MKL_INT *ja , MKL_INT
*desca , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcdttrf (MKL_INT *n , MKL_Complex8 *dl , MKL_Complex8 *d , MKL_Complex8 *du ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzdtttrf (MKL_INT *n , MKL_Complex16 *dl , MKL_Complex16 *d , MKL_Complex16 *du ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?dttrf` function computes the *LU* factorization of an *n*-by-*n* real/complex diagonally dominant-like tridiagonal distributed matrix $A(1:n, ja:ja+n-1)$ without pivoting for stability.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$$A(1:n, ja:ja+n-1) = P * L * U * P^T,$$

where *P* is a permutation matrix, and *L* and *U* are banded lower and upper triangular matrices, respectively.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

n	(global) The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$ ($n \geq 0$).
dl, d, du	(local) Pointers to the local arrays of size nb_a each. On entry, the array dl contains the local part of the global vector storing the subdiagonal elements of the matrix. Globally, $dl[0]$ is not referenced, and dl must be aligned with d . On entry, the array d contains the local part of the global vector storing the diagonal elements of the matrix. On entry, the array du contains the local part of the global vector storing the super-diagonal elements of the matrix. $du[n-1]$ is not referenced, and du must be aligned with d .
ja	(global) The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A . If $dtype_a = 501$, then $dlen_ \geq 7$; else if $dtype_a = 1$, then $dlen_ \geq 9$.
laf	(local) The size of the array af . Must be $laf \geq 2*(NB+2)$. If laf is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af[0]$.
$work$	(local) Same type as d . Workspace array of size $lwork$.
$lwork$	(local or global) The size of the $work$ array, must be at least $lwork \geq 8*NPCOL$.

Output Parameters

dl, d, du	On exit, overwritten by the information containing the factors of the matrix.
af	(local) Array of size laf . Auxiliary fill-in space. The fill-in space is created in a call to the factorization function <code>p?dttrf</code> and is stored in af .

Note that if a linear system is to be solved using `p?dttrs` after the factorization function, `af` must not be altered.

`work[0]`

On exit, `work[0]` contains the minimum value of `lwork` required for optimum performance.

`info`

(global)

If `info=0`, the execution is successful.

`info < 0`:

If the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then `info = -($i \times 100 + j$)`; if the i -th argument is a scalar and had an illegal value, then `info = - i` .

`info > 0`:

If `info = $k \leq$ NPROCS`, the submatrix stored on processor `info` and factored locally was not diagonally dominant-like, and the factorization was not completed.

If `info = $k >$ NPROCS`, the submatrix stored on processor `info-NPROCS` representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite distributed matrix.

Syntax

```
void pspotrf (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *info );

void pdpotrf (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *info );

void pcpotrf (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );

void pzpotrf (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?potrf` function computes the Cholesky factorization of a real symmetric or complex Hermitian positive-definite distributed n -by- n matrix $A(ia:ia+n-1, ja:ja+n-1)$, denoted below as $\text{sub}(A)$.

The factorization has the form

$\text{sub}(A) = U^H * U$ if `uplo='U'`, or

$\text{sub}(A) = L * L^H$ if `uplo='L'`

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

<i>uplo</i>	(global) Indicates whether the upper or lower triangular part of sub(<i>A</i>) is stored. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix sub(<i>A</i>) that is factored as $U^H * U$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix sub(<i>A</i>) that is factored as $L * L^H$.
<i>n</i>	(global) The order of the distributed matrix sub(<i>A</i>) ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size <i>lld_a</i> * <i>LOCc</i> (<i>ja</i> + <i>n</i> -1). On entry, this array contains the local pieces of the <i>n</i> -by- <i>n</i> symmetric/Hermitian distributed matrix sub(<i>A</i>) to be factored. Depending on <i>uplo</i> , the array <i>a</i> contains either the upper or the lower triangular part of the matrix sub(<i>A</i>) (see <i>uplo</i>).
<i>ia</i> , <i>ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by the Cholesky factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> .
<i>info</i>	(global) . If <i>info</i> =0, the execution is successful; <i>info</i> < 0: if the <i>i</i> -th argument is an array, and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . If <i>info</i> = <i>k</i> > 0, the leading minor of order <i>k</i> , <i>A</i> (<i>ia</i> : <i>ia</i> + <i>k</i> -1, <i>ja</i> : <i>ja</i> + <i>k</i> -1), is not positive-definite, and the factorization could not be completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite banded distributed matrix.

Syntax

```
void pspbtrf (char *uplo , MKL_INT *n , MKL_INT *bw , float *a , MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pdpbtrf (char *uplo , MKL_INT *n , MKL_INT *bw , double *a , MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );
```

```
void pcpbtrf (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_Complex8 *a , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzpbtrf (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_Complex16 *a , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?pbtrf` function computes the Cholesky factorization of an n -by- n real symmetric or complex Hermitian positive-definite banded distributed matrix $A(1:n, ja:ja+n-1)$.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$A(1:n, ja:ja+n-1) = P * U^H * U * P^T$, if `uplo='U'`, or

$A(1:n, ja:ja+n-1) = P * L * L^H * P^T$, if `uplo='L'`,

where P is a permutation matrix and U and L are banded upper and lower triangular matrices, respectively.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<code>uplo</code>	(global) Must be 'U' or 'L'. If <code>uplo = 'U'</code> , upper triangle of $A(1:n, ja:ja+n-1)$ is stored; If <code>uplo = 'L'</code> , lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<code>n</code>	(global) The order of the distributed submatrix $A(1:n, ja:ja+n-1)$. ($n \geq 0$).
<code>bw</code>	(global) The number of superdiagonals of the distributed matrix if <code>uplo = 'U'</code> , or the number of subdiagonals if <code>uplo = 'L'</code> ($bw \geq 0$).
<code>a</code>	(local) Pointer into the local memory to an array of size <code>ld_a*LOCc(ja+n-1)</code> . On entry, this array contains the local pieces of the upper or lower triangle of the symmetric/Hermitian band distributed matrix $A(1:n, ja:ja+n-1)$ to be factored.
<code>ja</code>	(global) The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).

<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>dtype_a</i> = 501, then <i>dlen_</i> ≥ 7; else if <i>dtype_a</i> = 1, then <i>dlen_</i> ≥ 9.
<i>laf</i>	(local) The size of the array <i>af</i> . Must be $laf \geq (NB+2*bw) * bw$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> [0].
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the <i>work</i> array, must be $lwork \geq bw^2$.

Output Parameters

<i>a</i>	On exit, if <i>info</i> =0, contains the permuted triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization of the band matrix $A(1:n, ja:ja+n-1)$, as specified by <i>uplo</i> .
<i>af</i>	(local) Array of size <i>laf</i> . Auxiliary fill-in space. The fill-in space is created in a call to the factorization function <i>p?pbtrf</i> and stored in <i>af</i> . Note that if a linear system is to be solved using <i>p?pbtrs</i> after the factorization function, <i>af</i> must not be altered.
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . <i>info</i> > 0: If <i>info</i> = <i>k</i> ≤ NPROCS, the submatrix stored on processor <i>info</i> and factored locally was not positive definite, and the factorization was not completed. If <i>info</i> = <i>k</i> > NPROCS, the submatrix stored on processor <i>info</i> -NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pttrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite tridiagonal distributed matrix.

Syntax

```
void pspttrf (MKL_INT *n , float *d , float *e , MKL_INT *ja , MKL_INT *desca , float
*af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pdpttrf (MKL_INT *n , double *d , double *e , MKL_INT *ja , MKL_INT *desca , double
*af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcpttrf (MKL_INT *n , float *d , MKL_Complex8 *e , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzpttrf (MKL_INT *n , double *d , MKL_Complex16 *e , MKL_INT *ja , MKL_INT
*desca , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT *lwork ,
MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?pttrf` function computes the Cholesky factorization of an n -by- n real symmetric or complex hermitian positive-definite tridiagonal distributed matrix $A(1:n, ja:ja+n-1)$.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$$A(1:n, ja:ja+n-1) = P * L * D * L^H * P^T, \text{ or}$$

$$A(1:n, ja:ja+n-1) = P * U^H * D * U * P^T,$$

where P is a permutation matrix, and U and L are tridiagonal upper and lower triangular matrices, respectively.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

n	(global) The order of the distributed submatrix $A(1:n, ja:ja+n-1)$ ($n \geq 0$).
d, e	(local) Pointers into the local memory to arrays of size <code>nb_a</code> each. On entry, the array d contains the local part of the global vector storing the main diagonal of the distributed matrix A .

On entry, the array *e* contains the local part of the global vector storing the upper diagonal of the distributed matrix *A*.

ja (global) The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *dtype_a* = 501, then *dlen_* ≥ 7;

else if *dtype_a* = 1, then *dlen_* ≥ 9.

laf (local) The size of the array *af*.

Must be *laf* ≥ *nb_a* + 2.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*[0].

work (local) Workspace array of size *lwork* .

lwork (local or global) The size of the *work* array, must be at least

lwork ≥ 8 * NPCOL.

Output Parameters

d, e On exit, overwritten by the details of the factorization.

af (local)

Array of size *laf*.

Auxiliary fill-in space. The fill-in space is created in a call to the factorization function *p?pttrf* and stored in *af*.

Note that if a linear system is to be solved using *p?pttrs* after the factorization function, *af* must not be altered.

work[0] On exit, *work*[0] contains the minimum value of *lwork* required for optimum performance.

info (global)

If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:

If *info* = *k* ≤ NPCOL, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.

If *info* = *k* > NPCOL, the submatrix stored on processor *info*-NPCOL representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Solving Systems of Linear Equations: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see [Routines for Matrix Factorization](#) in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

p?getrs

Solves a system of distributed linear equations with a general square matrix, using the LU factorization computed by p?getrf.

Syntax

```
void psgetrs (char *trans , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , float *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );

void pdgetrs (char *trans , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , double *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );

void pcgetrs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_INT *info );

void pzgetrs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?getrs function solves a system of distributed linear equations with a general n -by- n distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ using the LU factorization computed by p?getrf.

The system has one of the following forms specified by *trans*:

$\text{sub}(A)*X = \text{sub}(B)$ (no transpose),

$\text{sub}(A)^T*X = \text{sub}(B)$ (transpose),

$\text{sub}(A)^H*X = \text{sub}(B)$ (conjugate transpose),

where $\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+\text{nrhs}-1)$.

Before calling this function, you must call p?getrf to compute the LU factorization of $\text{sub}(A)$.

Input Parameters

<i>trans</i>	(global) Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $\text{sub}(A)*X = \text{sub}(B)$ is solved for X . If <i>trans</i> = 'T', then $\text{sub}(A)^T*X = \text{sub}(B)$ is solved for X .
--------------	---

If $trans = 'C'$, then $sub(A)^H * X = sub(B)$ is solved for X .

<i>n</i>	(global) The number of linear equations; the order of the matrix $sub(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) The number of right hand sides; the number of columns of the distributed matrix $sub(B)$ ($nrhs \geq 0$).
<i>a, b</i>	(local) Pointers into the local memory to arrays of local sizes $lld_a * LOCC(ja+n-1)$ and $lld_b * LOCC(jb+nrhs-1)$, respectively. On entry, the array <i>a</i> contains the local pieces of the factors <i>L</i> and <i>U</i> from the factorization $sub(A) = P * L * U$; the unit diagonal elements of <i>L</i> are not stored. On entry, the array <i>b</i> contains the right hand sides $sub(B)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix $sub(A)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>ipiv</i>	(local) Array of size of $LOCr(m_a) + mb_a$. Contains the pivoting information: local row <i>i</i> of the matrix was interchanged with the global row <i>ipiv</i> [<i>i</i> -1]. This array is tied to the distributed matrix <i>A</i> .
<i>ib, jb</i>	(global) The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the matrix $sub(B)$, respectively.
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>b</i>	On exit, overwritten by the solution distributed matrix <i>X</i> .
<i>info</i>	If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gbtrs

Solves a system of distributed linear equations with a general band matrix, using the LU factorization computed by p?gbtrf.

Syntax

```
void psgbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
float *a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , float *b , MKL_INT *ib ,
MKL_INT *descb , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT
*info );
```



```

void pdgbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
double *a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , double *b , MKL_INT *ib ,
MKL_INT *descb , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT
*info );

void pcgbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzgbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );

```

Include Files

- mkl_scalapack.h

Description

The `p?gbtrs` function solves a system of distributed linear equations with a general band distributed matrix $\text{sub}(A) = A(1:n, ja:ja+n-1)$ using the LU factorization computed by `p?gbtrf`.

The system has one of the following forms specified by `trans`:

$\text{sub}(A)*X = \text{sub}(B)$ (no transpose),

$\text{sub}(A)^T*X = \text{sub}(B)$ (transpose),

$\text{sub}(A)^H*X = \text{sub}(B)$ (conjugate transpose),

where $\text{sub}(B) = B(ib:ib+n-1, 1:nrhs)$.

Before calling this function, you must call `p?gbtrf` to compute the LU factorization of $\text{sub}(A)$.

Input Parameters

<code>trans</code>	(global) Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <code>trans</code> = 'N', then $\text{sub}(A)*X = \text{sub}(B)$ is solved for X . If <code>trans</code> = 'T', then $\text{sub}(A)^T*X = \text{sub}(B)$ is solved for X . If <code>trans</code> = 'C', then $\text{sub}(A)^H*X = \text{sub}(B)$ is solved for X .
<code>n</code>	(global) The number of linear equations; the order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<code>bwl</code>	(global) The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
<code>bwu</code>	(global) The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).
<code>nrhs</code>	(global) The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).
<code>a, b</code>	(local)

Pointers into the local memory to arrays of local sizes $lld_a * LOCC(ja+n-1)$ and $lld_b * LOCC(nrhs)$, respectively.

The array *a* contains details of the *LU* factorization of the distributed band matrix *A*.

On entry, the array *b* contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.

<i>ja</i>	(global) The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>dtype_a</i> = 501, then <i>dlen_</i> ≥ 7; else if <i>dtype_a</i> = 1, then <i>dlen_</i> ≥ 9.
<i>ib</i>	(global) The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>dtype_b</i> = 502, then <i>dlen_</i> ≥ 7; else if <i>dtype_b</i> = 1, then <i>dlen_</i> ≥ 9.
<i>laf</i>	(local) The size of the array <i>af</i> . Must be $laf \geq nb_a * (bwl + bwu) + 6 * (bwl + bwu) * (bwl + 2 * bwu)$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> [0].
<i>work</i>	(local) Same type as <i>a</i> . Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the <i>work</i> array, must be at least $lwork \geq nrhs * (nb_a + 2 * bwl + 4 * bwu)$.

Output Parameters

<i>ipiv</i>	(local) array. The size of <i>ipiv</i> must be ≥ <i>nb_a</i> . Contains pivot indices for local factorizations. Note that you should not alter the contents of this array between factorization and solve.
<i>b</i>	On exit, overwritten by the local pieces of the solution distributed matrix <i>X</i> .
<i>af</i>	(local) Array of size <i>laf</i> . Auxiliary Fill-in space. The fill-in space is created in a call to the factorization function <i>p?gbtrf</i> and is stored in <i>af</i> . Note that if a linear system is to be solved using <i>p?gbtrs</i> after the factorization function, <i>af</i> must not be altered after the factorization.

`work[0]` On exit, `work[0]` contains the minimum value of `lwork` required for optimum performance.

`info` If `info=0`, the execution is successful.

`info < 0`:

If the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then `info = -(i*100+j)`; if the i -th argument is a scalar and had an illegal value, then `info = -i`.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dbtrs

Solves a system of linear equations with a diagonally dominant-like banded distributed matrix using the factorization computed by `p?dbtrf`.

Syntax

```
void psdbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
float *a , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb ,
float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pddbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
double *a , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb ,
double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcdbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzdbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- `mk1_scalapack.h`

Description

The `p?dbtrs` function solves for X one of the systems of equations:

$\text{sub}(A) * X = \text{sub}(B)$,
 $(\text{sub}(A))^T * X = \text{sub}(B)$, or
 $(\text{sub}(A))^H * X = \text{sub}(B)$,

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is a diagonally dominant-like banded distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This function uses the LU factorization computed by `p?dbtrf`.

Input Parameters

`trans` (global) Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If $trans = 'N'$, then $sub(A)*X = sub(B)$ is solved for X .

If $trans = 'T'$, then $(sub(A))^T * X = sub(B)$ is solved for X .

If $trans = 'C'$, then $(sub(A))^H * X = sub(B)$ is solved for X .

n	(global) The order of the distributed matrix $sub(A)$ ($n \geq 0$).
bwl	(global) The number of subdiagonals within the band of A ($0 \leq bwl \leq n-1$).
bwu	(global) The number of superdiagonals within the band of A ($0 \leq bwu \leq n-1$).
$nrhs$	(global) The number of right hand sides; the number of columns of the distributed matrix $sub(B)$ ($nrhs \geq 0$).
a, b	(local) Pointers into the local memory to arrays of local sizes $lld_a*LOCc(ja+n-1)$ and $lld_b*LOCc(nrhs)$, respectively. On entry, the array a contains details of the LU factorization of the band matrix A , as computed by <code>p?dbtrf</code> . On entry, the array b contains the local pieces of the right hand side distributed matrix $sub(B)$.
ja	(global) The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A . If $dtype_a = 501$, then $dlen_ \geq 7$; else if $dtype_a = 1$, then $dlen_ \geq 9$.
ib	(global) The row index in the global matrix B indicating the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).
$descb$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix B . If $dtype_b = 502$, then $dlen_ \geq 7$; else if $dtype_b = 1$, then $dlen_ \geq 9$.
$af, work$	(local) Arrays of size laf and $lwork$, respectively The array af contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function <code>p?dbtrf</code> and is stored in af . The array $work$ is a workspace array.
laf	(local) The size of the array af . Must be $laf \geq NB * (bwl + bwu) + 6 * (\max(bwl, bwu))^2$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*[0].

lwork (local or global) The size of the array *work*, must be at least
 $lwork \geq (\max(bwl, bwu))^2$.

Output Parameters

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

work[0] On exit, *work*[0] contains the minimum value of *lwork* required for optimum performance.

info If *info*=0, the execution is successful. *info* < 0:
 If the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dttrs

Solves a system of linear equations with a diagonally dominant-like tridiagonal distributed matrix using the factorization computed by p?dttrf.

Syntax

```
void psdttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , float *dl , float *d , float
*du , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float
*af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pddttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , double *dl , double *d , double
*du , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double
*af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcdttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *dl ,
MKL_Complex8 *d , MKL_Complex8 *du , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzdttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *dl ,
MKL_Complex16 *d , MKL_Complex16 *du , MKL_INT *ja , MKL_INT *desca , MKL_Complex16
*b , MKL_INT *ib , MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16
*work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The *p?dttrs* function solves for *X* one of the systems of equations:

$\text{sub}(A) * X = \text{sub}(B),$
 $(\text{sub}(A))^T * X = \text{sub}(B),$ or

$$(\text{sub}(A))^H * X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is a diagonally dominant-like tridiagonal distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This function uses the *LU* factorization computed by [p?dttrf](#).

Input Parameters

<i>trans</i>	<p>(global) Must be 'N' or 'T' or 'C'.</p> <p>Indicates the form of the equations:</p> <p>If <i>trans</i> = 'N', then $\text{sub}(A) * X = \text{sub}(B)$ is solved for <i>X</i>.</p> <p>If <i>trans</i> = 'T', then $(\text{sub}(A))^T * X = \text{sub}(B)$ is solved for <i>X</i>.</p> <p>If <i>trans</i> = 'C', then $(\text{sub}(A))^H * X = \text{sub}(B)$ is solved for <i>X</i>.</p>
<i>n</i>	(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>dl, d, du</i>	<p>(local)</p> <p>Pointers to the local arrays of size <i>nb_a</i> each.</p> <p>On entry, these arrays contain details of the factorization. Globally, <i>dl</i>[0] and <i>du</i>[<i>n</i>-1] are not referenced; <i>dl</i> and <i>du</i> must be aligned with <i>d</i>.</p>
<i>ja</i>	(global) The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p> <p>If <i>dtype_a</i> = 501 or <i>dtype_a</i> = 502, then <i>dlen_</i> ≥ 7;</p> <p>else if <i>dtype_a</i> = 1, then <i>dlen_</i> ≥ 9.</p>
<i>b</i>	<p>(local) Same type as <i>d</i>.</p> <p>Pointer into the local memory to an array of local size <i>ld_b</i>*<i>LOCc</i>(<i>nrhs</i>)</p> <p>On entry, the array <i>b</i> contains the local pieces of the <i>n</i>-by-<i>nrhs</i> right hand side distributed matrix $\text{sub}(B)$.</p>
<i>ib</i>	(global) The row index in the global matrix <i>B</i> indicating the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>B</i>.</p> <p>If <i>dtype_b</i> = 502, then <i>dlen_</i> ≥ 7;</p> <p>else if <i>dtype_b</i> = 1, then <i>dlen_</i> ≥ 9.</p>
<i>af, work</i>	<p>(local)</p> <p>Arrays of size <i>laf</i> and (<i>lwork</i>), respectively.</p>

The array *af* contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function `p?dttrf` and is stored in *af*. If a linear system is to be solved using `p?dttrs` after the factorization function, *af* must not be altered.

The array *work* is a workspace array.

laf

(local) The size of the array *af*.

Must be $laf \geq NB * (bwl + bwu) + 6 * (bwl + bwu) * (bwl + 2 * bwu)$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*[0].

lwork

(local or global) The size of the array *work*, must be at least $lwork \geq 10 * NPCOL + 4 * nrhs$.

Output Parameters

b

On exit, this array contains the local pieces of the solution distributed matrix *X*.

work[0]

On exit, *work*[0] contains the minimum value of *lwork* required for optimum performance.

info

If *info*=0, the execution is successful. *info* < 0:

If the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?potrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian distributed positive-definite matrix.

Syntax

```
void pspotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT
*info );
```

```
void pdpotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_INT *info );
```

```
void pcpotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
```

```
void pzpotsrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?potrs` function solves for X a system of distributed linear equations in the form:

$$\text{sub}(A) * X = \text{sub}(B),$$

where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ is an n -by- n real symmetric or complex Hermitian positive definite distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+nrhs-1)$.

This function uses Cholesky factorization

$$\text{sub}(A) = U^H * U, \text{ or } \text{sub}(A) = L * L^H$$

computed by `p?potrf`.

Input Parameters

<i>uplo</i>	(global) Must be 'U' or 'L'. If <i>uplo</i> = 'U', upper triangle of $\text{sub}(A)$ is stored; If <i>uplo</i> = 'L', lower triangle of $\text{sub}(A)$ is stored.
<i>n</i>	(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>a, b</i>	(local) Pointers into the local memory to arrays of local sizes $\text{lld_a} * \text{LOCc}(\text{ja}+n-1)$ and $\text{lld_b} * \text{LOCc}(\text{jb}+nrhs-1)$, respectively. The array <i>a</i> contains the factors L or U from the Cholesky factorization $\text{sub}(A) = L * L^H$ or $\text{sub}(A) = U^H * U$, as computed by <code>p?potrf</code> . On entry, the array <i>b</i> contains the local pieces of the right hand sides $\text{sub}(B)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>ib, jb</i>	(global) The row and column indices in the global matrix B indicating the first row and the first column of the matrix $\text{sub}(B)$, respectively.
<i>descb</i>	(local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix B .

Output Parameters

<i>b</i>	Overwritten by the local pieces of the solution matrix X .
<i>info</i>	If <i>info</i> =0, the execution is successful. <i>info</i> < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian positive-definite band matrix.

Syntax

```
void pspbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , float *a , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *af , MKL_INT
*laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pdpbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , double *a ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *af ,
MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcpbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , MKL_Complex8 *a ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzpbttrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , MKL_Complex16
*a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- mkl_scalapack.h

Description

The `p?pbtrs` function solves for X a system of distributed linear equations in the form:

$$\text{sub}(A) * X = \text{sub}(B) ,$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an n -by- n real symmetric or complex Hermitian positive definite distributed band matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This function uses Cholesky factorization

$$\text{sub}(A) = P * U^H * U * P^T, \text{ or } \text{sub}(A) = P * L * L^H * P^T$$

computed by `p?pbtrf`.

Input Parameters

<code>uplo</code>	(global) Must be 'U' or 'L'. If <code>uplo</code> = 'U', upper triangle of $\text{sub}(A)$ is stored; If <code>uplo</code> = 'L', lower triangle of $\text{sub}(A)$ is stored.
<code>n</code>	(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<code>bw</code>	(global) The number of superdiagonals of the distributed matrix if <code>uplo</code> = 'U', or the number of subdiagonals if <code>uplo</code> = 'L' ($bw \geq 0$).
<code>nrhs</code>	(global) The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).

<i>a, b</i>	<p>(local)</p> <p>Pointers into the local memory to arrays of local sizes $lld_a * LOCc(ja+n-1)$ and $lld_b * LOCc(nrhs-1)$, respectively.</p> <p>The array <i>a</i> contains the permuted triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $sub(A) = P * U^H * U * P^T$, or $sub(A) = P * L * L^H * P^T$ of the band matrix <i>A</i>, as returned by <code>p?pbtrf</code>.</p> <p>On entry, the array <i>b</i> contains the local pieces of the <i>n</i>-by-<i>nrhs</i> right hand side distributed matrix $sub(B)$.</p>
<i>ja</i>	(global) The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p> <p>If <i>dtype_a</i> = 501, then <i>dlen_</i> ≥ 7; else if <i>dtype_a</i> = 1, then <i>dlen_</i> ≥ 9.</p>
<i>ib</i>	(global) The row index in the global matrix <i>B</i> indicating the first row of the matrix $sub(B)$.
<i>descb</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>B</i>.</p> <p>If <i>dtype_b</i> = 502, then <i>dlen_</i> ≥ 7; else if <i>dtype_b</i> = 1, then <i>dlen_</i> ≥ 9.</p>
<i>af, work</i>	<p>(local) Arrays, same type as <i>a</i>.</p> <p>The array <i>af</i> is of size <i>laf</i>. It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function <code>p?dbtrf</code> and is stored in <i>af</i>.</p> <p>The array <i>work</i> is a workspace array of size <i>lwork</i>.</p>
<i>laf</i>	<p>(local) The size of the array <i>af</i>.</p> <p>Must be $laf \geq nrhs * bw$.</p> <p>If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i>[0].</p>
<i>lwork</i>	(local or global) The size of the array <i>work</i> , must be at least $lwork \geq bw^2$.

Output Parameters

<i>b</i>	On exit, if <i>info</i> =0, this array contains the local pieces of the <i>n</i> -by- <i>nrhs</i> solution distributed matrix <i>X</i> .
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	<p>If <i>info</i>=0, the execution is successful.</p> <p><i>info</i> < 0:</p>

If the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then `info = -(i*100+j)`; if the i -th argument is a scalar and had an illegal value, then `info = -i`.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal distributed matrix using the factorization computed by p?pttrf.

Syntax

```
void pspttrs (MKL_INT *n , MKL_INT *nrhs , float *d , float *e , MKL_INT *ja , MKL_INT
*desca , float *b , MKL_INT *ib , MKL_INT *descb , float *af , MKL_INT *laf , float
*work , MKL_INT *lwork , MKL_INT *info );

void pdpttrs (MKL_INT *n , MKL_INT *nrhs , double *d , double *e , MKL_INT *ja , MKL_INT
*desca , double *b , MKL_INT *ib , MKL_INT *descb , double *af , MKL_INT *laf , double
*work , MKL_INT *lwork , MKL_INT *info );

void pcpttrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *d , MKL_Complex8 *e ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzpttrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *d , MKL_Complex16 *e ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- `mk1_scalapack.h`

Description

The `p?pttrs` function solves for X a system of distributed linear equations in the form:

$$\text{sub}(A) * X = \text{sub}(B) ,$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an n -by- n real symmetric or complex Hermitian positive definite tridiagonal distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This function uses the factorization

$$\text{sub}(A) = P * L * D * L^H * P^T, \text{ or } \text{sub}(A) = P * U^H * D * U * P^T$$

computed by `p?pttrf`.

Input Parameters

`uplo` (global, used in complex flavors only)
 Must be 'U' or 'L'.
 If `uplo = 'U'`, upper triangle of $\text{sub}(A)$ is stored;
 If `uplo = 'L'`, lower triangle of $\text{sub}(A)$ is stored.

<i>n</i>	(global) The order of the distributed matrix sub(<i>A</i>) ($n \geq 0$).
<i>nrhs</i>	(global) The number of right hand sides; the number of columns of the distributed matrix sub(<i>B</i>) ($nrhs \geq 0$).
<i>d, e</i>	(local) Pointers into the local memory to arrays of size <i>nb_a</i> each. These arrays contain details of the factorization as returned by <code>p?pttrf</code>
<i>ja</i>	(global) The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>dtype_a</i> = 501 or <i>dtype_a</i> = 502, then $dlen_ \geq 7$; else if <i>dtype_a</i> = 1, then $dlen_ \geq 9$.
<i>b</i>	(local) Same type as <i>d, e</i> . Pointer into the local memory to an array of local size <i>lld_b</i> * <i>LOCc</i> (<i>nrhs</i>). On entry, the array <i>b</i> contains the local pieces of the <i>n</i> -by- <i>nrhs</i> right hand side distributed matrix sub(<i>B</i>).
<i>ib</i>	(global) The row index in the global matrix <i>B</i> indicating the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> . If <i>dtype_b</i> = 502, then $dlen_ \geq 7$; else if <i>dtype_b</i> = 1, then $dlen_ \geq 9$.
<i>af, work</i>	(local) Arrays of size <i>laf</i> and (<i>lwork</i>), respectively. The array <i>af</i> contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function <code>p?pttrf</code> and is stored in <i>af</i> . The array <i>work</i> is a workspace array.
<i>laf</i>	(local) The size of the array <i>af</i> . Must be $laf \geq nb_a + 2$. If <i>laf</i> is not large enough, an error code is returned and the minimum acceptable size will be returned in <i>af</i> [0].
<i>lwork</i>	(local or global) The size of the array <i>work</i> , must be at least $lwork \geq (10 + 2 * \min(100, nrhs)) * NPCOL + 4 * nrhs$.

Output Parameters

<i>b</i>	On exit, this array contains the local pieces of the solution distributed matrix <i>X</i> .
----------	---

`work[0]` On exit, `work[0]` contains the minimum value of `lwork` required for optimum performance.

`info` If `info=0`, the execution is successful.
`info < 0`:
 if the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then `info = -(i*100+j)`; if the *i*-th argument is a scalar and had an illegal value, then `info = -i`.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trtrs

Solves a system of linear equations with a triangular distributed matrix.

Syntax

```
void pstrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );

void pdtrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_INT *info );

void pctrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );

void pztrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?trtrs` function solves for X one of the following systems of linear equations:

$\text{sub}(A) * X = \text{sub}(B)$,
 $(\text{sub}(A))^T * X = \text{sub}(B)$, or
 $(\text{sub}(A))^H * X = \text{sub}(B)$,

where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ is a triangular distributed matrix of order n , and $\text{sub}(B)$ denotes the distributed matrix $B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+\text{nrhs}-1)$.

A check is made to verify that $\text{sub}(A)$ is nonsingular.

Input Parameters

`uplo` (global) Must be 'U' or 'L'.
 Indicates whether $\text{sub}(A)$ is upper or lower triangular:
 If `uplo = 'U'`, then $\text{sub}(A)$ is upper triangular.

	If <i>uplo</i> = 'L', then sub(<i>A</i>) is lower triangular.
<i>trans</i>	(global) Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then sub(<i>A</i>)* <i>X</i> = sub(<i>B</i>) is solved for <i>X</i> . If <i>trans</i> = 'T', then sub(<i>A</i>) ^T * <i>X</i> = sub(<i>B</i>) is solved for <i>X</i> . If <i>trans</i> = 'C', then sub(<i>A</i>) ^H * <i>X</i> = sub(<i>B</i>) is solved for <i>X</i> .
<i>diag</i>	(global) Must be 'N' or 'U'. If <i>diag</i> = 'N', then sub(<i>A</i>) is not a unit triangular matrix. If <i>diag</i> = 'U', then sub(<i>A</i>) is unit triangular.
<i>n</i>	(global) The order of the distributed matrix sub(<i>A</i>) (<i>n</i> ≥0).
<i>nrhs</i>	(global) The number of right-hand sides; i.e., the number of columns of the distributed matrix sub(<i>B</i>) (<i>nrhs</i> ≥0).
<i>a, b</i>	(local) Pointers into the local memory to arrays of local sizes <i>lld_a</i> * <i>LOCc</i> (<i>ja</i> + <i>n</i> -1) and <i>lld_b</i> * <i>LOCc</i> (<i>jb</i> + <i>nrhs</i> -1), respectively. The array <i>a</i> contains the local pieces of the distributed triangular matrix sub(<i>A</i>). If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of sub(<i>A</i>) contains the upper triangular matrix, and the strictly lower triangular part of sub(<i>A</i>) is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular matrix, and the strictly upper triangular part of sub(<i>A</i>) is not referenced. If <i>diag</i> = 'U', the diagonal elements of sub(<i>A</i>) are also not referenced and are assumed to be 1. On entry, the array <i>b</i> contains the local pieces of the right hand side distributed matrix sub(<i>B</i>).
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>ib, jb</i>	(global) The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the matrix sub(<i>B</i>), respectively.
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>b</i>	On exit, if <i>info</i> =0, sub(<i>B</i>) is overwritten by the solution matrix <i>X</i> .
<i>info</i>	If <i>info</i> =0, the execution is successful.

info < 0:

if the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:

if *info* = *i*, the *i*-th diagonal element of sub(*A*) is zero, indicating that the submatrix is singular and the solutions *X* have not been computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Estimating the Condition Number: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for estimating the condition number of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations. Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

p?gecon

Estimates the reciprocal of the condition number of a general distributed matrix in either the 1-norm or the infinity-norm.

Syntax

```
void psgecon (char *norm , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *anorm , float *rcond , float *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *liwork , MKL_INT *info );
```

```
void pdgecon (char *norm , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *anorm , double *rcond , double *work , MKL_INT *lwork , MKL_INT
*iwork , MKL_INT *liwork , MKL_INT *info );
```

```
void pcgecon (char *norm , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *anorm , float *rcond , MKL_Complex8 *work , MKL_INT *lwork ,
float *rwork , MKL_INT *lrwork , MKL_INT *info );
```

```
void pzgecon (char *norm , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *anorm , double *rcond , MKL_Complex16 *work , MKL_INT *lwork ,
double *rwork , MKL_INT *lrwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?gecon function estimates the reciprocal of the condition number of a general distributed real/complex matrix sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*) in either the 1-norm or infinity-norm, using the *LU* factorization computed by p?getrf.

An estimate is obtained for ||(sub(*A*))⁻¹||, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

<i>norm</i>	<p>(global) Must be '1' or 'O' or 'I'.</p> <p>Specifies whether the 1-norm condition number or the infinity-norm condition number is required.</p> <p>If <i>norm</i> = '1' or 'O', then the 1-norm is used;</p> <p>If <i>norm</i> = 'I', then the infinity-norm is used.</p>
<i>n</i>	(global) The order of the distributed matrix sub(A) ($n \geq 0$).
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size <i>ld_a</i>*<i>LOCc</i>(<i>ja</i>+<i>n</i>-1).</p> <p>The array <i>a</i> contains the local pieces of the factors <i>L</i> and <i>U</i> from the factorization $sub(A) = P*L*U$; the unit diagonal elements of <i>L</i> are not stored.</p>
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix sub(A), respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>anorm</i>	<p>(global)</p> <p>If <i>norm</i> = '1' or 'O', the 1-norm of the original distributed matrix sub(A);</p> <p>If <i>norm</i> = 'I', the infinity-norm of the original distributed matrix sub(A).</p>
<i>work</i>	<p>(local)</p> <p>The array <i>work</i> of size <i>lwork</i> is a workspace array.</p>
<i>lwork</i>	<p>(local or global) The size of the array <i>work</i>.</p> <p>For real flavors:</p> <p><i>lwork</i> must be at least</p> $lwork \geq 2*LOCr(n+\text{mod}(ia-1, mb_a))+2*LOCc(n+\text{mod}(ja-1, nb_a))+\max(2, \max(nb_a*\max(1, \text{iceil}(\text{NPROW}-1, \text{NPCOL})), LOCc(n+\text{mod}(ja-1, nb_a)) + nb_a*\max(1, \text{iceil}(\text{NPCOL}-1, \text{NPROW}))))).$ <p>For complex flavors:</p> <p><i>lwork</i> must be at least</p> $lwork \geq 2*LOCr(n+\text{mod}(ia-1, mb_a))+\max(2, \max(nb_a*\text{iceil}(\text{NPROW}-1, \text{NPCOL}), LOCc(n+\text{mod}(ja-1, nb_a))+nb_a*\text{iceil}(\text{NPCOL}-1, \text{NPROW}))).$

LOCr and *LOCc* values can be computed using the ScaLAPACK tool function `numroc`; *NPROW* and *NPCOL* can be determined by calling the function `blacs_gridinfo`.

NOTE

`iceil(x,y)` is the ceiling of x/y , and `mod(x,y)` is the integer remainder of x/y .

<i>iwork</i>	(local) Workspace array of size <i>liwork</i> . Used in real flavors only.
<i>liwork</i>	(local or global) The size of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ia-1, mb_a))$
<i>rwork</i>	(local) Workspace array of size <i>lrwork</i> . Used in complex flavors only.
<i>lrwork</i>	(local or global) The size of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq \max(1, 2 * LOCc(n + \text{mod}(ja-1, nb_a)))$

Output Parameters

<i>rcond</i>	(global) The reciprocal of the condition number of the distributed matrix sub(A). See Description.
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> [0]	On exit, <i>iwork</i> [0] contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> [0]	On exit, <i>rwork</i> [0] contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) If <i>info</i> =0, the execution is successful. $info < 0:$ <p>If the <i>i</i>-th argument is an array and the <i>j</i>-th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = -(<i>i</i>*100+<i>j</i>); if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = -<i>i</i>.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pocon

Estimates the reciprocal of the condition number (in the 1 - norm) of a symmetric / Hermitian positive-definite distributed matrix.

Syntax

```
void psपोcon (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *anorm , float *rcond , float *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *liwork , MKL_INT *info );

void pdपोcon (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *anorm , double *rcond , double *work , MKL_INT *lwork , MKL_INT
*iwork , MKL_INT *liwork , MKL_INT *info );

void pcपोcon (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *anorm , float *rcond , MKL_Complex8 *work , MKL_INT *lwork ,
float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pzपोcon (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *anorm , double *rcond , MKL_Complex16 *work , MKL_INT *lwork ,
double *rwork , MKL_INT *lrwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?पोcon function estimates the reciprocal of the condition number (in the 1 - norm) of a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$, using the Cholesky factorization $\text{sub}(A) = U^H * U$ or $\text{sub}(A) = L * L^H$ computed by p?पोtrf.

An estimate is obtained for $\|(\text{sub}(A))^{-1}\|$, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|\text{sub}(A)\| \times \|(\text{sub}(A))^{-1}\|}$$

Input Parameters

<i>uplo</i>	(global) Must be 'U' or 'L'. Specifies whether the factor stored in $\text{sub}(A)$ is upper or lower triangular. If <i>uplo</i> = 'U', $\text{sub}(A)$ stores the upper triangular factor U of the Cholesky factorization $\text{sub}(A) = U^H * U$. If <i>uplo</i> = 'L', $\text{sub}(A)$ stores the lower triangular factor L of the Cholesky factorization $\text{sub}(A) = L * L^H$.
<i>n</i>	(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size $\text{ld}_a * \text{LOCc}(\text{ja}+n-1)$. The array <i>a</i> contains the local pieces of the factors L or U from the Cholesky factorization $\text{sub}(A) = U^H * U$, or $\text{sub}(A) = L * L^H$, as computed by p?पोtrf.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.
<i>anorm</i>	(global) The 1-norm of the symmetric/Hermitian distributed matrix sub(A).
<i>work</i>	(local) The array <i>work</i> of size <i>lwork</i> is a workspace array.
<i>lwork</i>	(local or global) The size of the array <i>work</i> . For real flavors: <i>lwork</i> must be at least $lwork \geq 2*LOCr(n+mod(ia-1,mb_a))+2*LOCc(n+mod(ja-1,nb_a))+max(2, max(nb_a*iceil(NPROW-1, NPCOL), LOCc(n+mod(ja-1,nb_a))+nb_a*iceil(NPCOL-1, NPROW)))$ For complex flavors: <i>lwork</i> must be at least $lwork \geq 2*LOCr(n+mod(ia-1,mb_a))+max(2, max(nb_a*max(1,iceil(NPROW-1, NPCOL)), LOCc(n+mod(ja-1,nb_a))+nb_a*max(1,iceil(NPCOL-1, NPROW))))$ If <i>lwork</i> = -1, then <i>lwork</i> is a global input and a workspace query is assumed. The routine only calculates the minimum and optimal size for all work arrays. Each value is returned in the first entry of the corresponding work array, and no error message is issued by <i>p_xerbla</i> .

NOTE

iceil(*x*,*y*) is the ceiling of *x*/*y*, and *mod*(*x*,*y*) is the integer remainder of *x*/*y*.

<i>iwork</i>	(local) Workspace array of size <i>liwork</i> . Used in real flavors only.
<i>liwork</i>	(local or global) The size of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n+mod(ia-1,mb_a))$. If <i>liwork</i> = -1, then <i>liwork</i> is a global input and a workspace query is assumed. The routine only calculates the minimum and optimal size for all work arrays. Each value is returned in the first entry of the corresponding work array, and no error message is issued by <i>p_xerbla</i> .
<i>rwork</i>	(local) Workspace array of size <i>lrwork</i> . Used in complex flavors only.
<i>lrwork</i>	(local or global) The size of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq 2*LOCc(n+mod(ja-1,nb_a))$. If <i>lrwork</i> = -1, then <i>lrwork</i> is a global input and a workspace query is assumed. The routine only calculates the minimum and optimal size for all work arrays. Each value is returned in the first entry of the corresponding work array, and no error message is issued by <i>p_xerbla</i> .

Output Parameters

<code>rcond</code>	(global) The reciprocal of the condition number of the distributed matrix sub(A).
<code>work[0]</code>	On exit, <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>iwork[0]</code>	On exit, <code>iwork[0]</code> contains the minimum value of <code>liwork</code> required for optimum performance (for real flavors).
<code>rwork[0]</code>	On exit, <code>rwork[0]</code> contains the minimum value of <code>lrwork</code> required for optimum performance (for complex flavors).
<code>info</code>	(global) If <code>info=0</code> , the execution is successful. <code>info < 0</code> : If the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then <code>info = -(i*100+j)</code> ; if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trcon

Estimates the reciprocal of the condition number of a triangular distributed matrix in either 1-norm or infinity-norm.

Syntax

```
void pstrcon (char *norm , char *uplo , char *diag , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *rcond , float *work , MKL_INT *lwork ,
MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pdtrcon (char *norm , char *uplo , char *diag , MKL_INT *n , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *rcond , double *work , MKL_INT *lwork ,
MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pctrcon (char *norm , char *uplo , char *diag , MKL_INT *n , MKL_Complex8 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *rcond , MKL_Complex8 *work ,
MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pztrcon (char *norm , char *uplo , char *diag , MKL_INT *n , MKL_Complex16 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *rcond , MKL_Complex16 *work ,
MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );
```

Include Files

- `mk1_scalapack.h`

Description

The `p?trcon` function estimates the reciprocal of the condition number of a triangular distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$, in either the 1-norm or the infinity-norm.

The norm of $\text{sub}(A)$ is computed and an estimate is obtained for $\|(\text{sub}(A))^{-1}\|$, then the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|sub(A)\| \times \|(sub(A))^{-1}\|}$$

Input Parameters

<i>norm</i>	<p>(global) Must be '1' or 'O' or 'I'.</p> <p>Specifies whether the 1-norm condition number or the infinity-norm condition number is required.</p> <p>If <i>norm</i> = '1' or 'O', then the 1-norm is used;</p> <p>If <i>norm</i> = 'I', then the infinity-norm is used.</p>
<i>uplo</i>	<p>(global) Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', sub(A) is upper triangular. If <i>uplo</i> = 'L', sub(A) is lower triangular.</p>
<i>diag</i>	<p>(global) Must be 'N' or 'U'.</p> <p>If <i>diag</i> = 'N', sub(A) is non-unit triangular. If <i>diag</i> = 'U', sub(A) is unit triangular.</p>
<i>n</i>	(global) The order of the distributed matrix sub(A), ($n \geq 0$).
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size <code>lld_a*LOCc(ja+n-1)</code>.</p> <p>The array <i>a</i> contains the local pieces of the triangular distributed matrix sub(A).</p> <p>If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of this distributed matrix contains the upper triangular matrix, and its strictly lower triangular part is not referenced.</p> <p>If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of this distributed matrix contains the lower triangular matrix, and its strictly upper triangular part is not referenced.</p> <p>If <i>diag</i> = 'U', the diagonal elements of sub(A) are also not referenced and are assumed to be 1.</p>
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.
<i>work</i>	<p>(local)</p> <p>The array <i>work</i> of size <i>lwork</i> is a workspace array.</p>
<i>lwork</i>	<p>(local or global) The size of the array <i>work</i>.</p> <p>For real flavors:</p>

lwork must be at least

$$lwork \geq 2 * LOCr(n + \text{mod}(ia - 1, mb_a)) + LOCc(n + \text{mod}(ja - 1, nb_a)) + \max(2, \max(nb_a * \max(1, \text{iceil}(\text{NPROW} - 1, \text{NPCOL})), LOCc(n + \text{mod}(ja - 1, nb_a)) + nb_a * \max(1, \text{iceil}(\text{NPCOL} - 1, \text{NPROW}))))).$$

For complex flavors:

lwork must be at least

$$lwork \geq 2 * LOCr(n + \text{mod}(ia - 1, mb_a)) + \max(2, \max(nb_a * \text{iceil}(\text{NPROW} - 1, \text{NPCOL}), LOCc(n + \text{mod}(ja - 1, nb_a)) + nb_a * \text{iceil}(\text{NPCOL} - 1, \text{NPROW}))).$$

NOTE

$\text{iceil}(x, y)$ is the ceiling of x/y , and $\text{mod}(x, y)$ is the integer remainder of x/y .

<i>iwork</i>	(local) Workspace array of size <i>liwork</i> . Used in real flavors only.
<i>liwork</i>	(local or global) The size of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ia - 1, mb_a)).$
<i>rwork</i>	(local) Workspace array of size <i>lrwork</i> . Used in complex flavors only.
<i>lrwork</i>	(local or global) The size of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq LOCc(n + \text{mod}(ja - 1, nb_a)).$

Output Parameters

<i>rcond</i>	(global) The reciprocal of the condition number of the distributed matrix sub(A).
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> [0]	On exit, <i>iwork</i> [0] contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> [0]	On exit, <i>rwork</i> [0] contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Refining the Solution and Estimating Its Error: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see [Routines for Matrix Factorization](#) and [Solving Systems of Linear Equations](#)).

p?gerfs

Improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

Syntax

```
void psgerfs (char *trans , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT
*descaf , MKL_INT *ipiv , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float
*x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *ferr , float *berr , float
*work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pdgerfs (char *trans , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT
*descaf , MKL_INT *ipiv , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *ferr , double *berr ,
double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pcgerfs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *iaf , MKL_INT *jaf ,
MKL_INT *descaf , MKL_INT *ipiv , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *ferr ,
float *berr , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *lrwork ,
MKL_INT *info );

void pzgerfs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf , MKL_INT *jaf ,
MKL_INT *descaf , MKL_INT *ipiv , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double
*ferr , double *berr , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT
*lrwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?gerfs function improves the computed solution to one of the systems of linear equations

$$\text{sub}(A) * \text{sub}(X) = \text{sub}(B),$$

$$\text{sub}(A)^T * \text{sub}(X) = \text{sub}(B), \text{ or}$$

$$\text{sub}(A)^H * \text{sub}(X) = \text{sub}(B) \text{ and provides error bounds and backward error estimates for the solution.}$$

Here $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$, $\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+nrhs-1)$, and $\text{sub}(X) = X(\text{ix}:\text{ix}+n-1, \text{jx}:\text{jx}+nrhs-1)$.

Input Parameters

<i>trans</i>	<p>(global) Must be 'N' or 'T' or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <i>trans</i> = 'N', the system has the form $\text{sub}(A) * \text{sub}(X) = \text{sub}(B)$ (No transpose);</p> <p>If <i>trans</i> = 'T', the system has the form $\text{sub}(A)^T * \text{sub}(X) = \text{sub}(B)$ (Transpose);</p> <p>If <i>trans</i> = 'C', the system has the form $\text{sub}(A)^H * \text{sub}(X) = \text{sub}(B)$ (Conjugate transpose).</p>
<i>n</i>	(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) The number of right-hand sides, i.e., the number of columns of the matrices $\text{sub}(B)$ and $\text{sub}(X)$ ($nrhs \geq 0$).
<i>a, af, b, x</i>	<p>(local)</p> <p>Pointers into the local memory to arrays of local sizes</p> <p><i>a</i>: $\text{lld_a} * \text{LOCc}(ja+n-1)$,</p> <p><i>af</i>: $\text{lld_af} * \text{LOCc}(jaf+n-1)$,</p> <p><i>b</i>: $\text{lld_b} * \text{LOCc}(jb+nrhs-1)$,</p> <p><i>x</i>: $\text{lld_x} * \text{LOCc}(jx+nrhs-1)$.</p> <p>The array <i>a</i> contains the local pieces of the distributed matrix $\text{sub}(A)$.</p> <p>The array <i>af</i> contains the local pieces of the distributed factors of the matrix $\text{sub}(A) = P * L * U$ as computed by p?getrf.</p> <p>The array <i>b</i> contains the local pieces of the distributed matrix of right hand sides $\text{sub}(B)$.</p> <p>On entry, the array <i>x</i> contains the local pieces of the distributed solution matrix $\text{sub}(X)$.</p>
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>iaf, jaf</i>	(global) The row and column indices in the global matrix <i>AF</i> indicating the first row and the first column of the matrix $\text{sub}(AF)$, respectively.
<i>descaf</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>AF</i> .
<i>ib, jb</i>	(global) The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the matrix $\text{sub}(B)$, respectively.
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>ix, jx</i>	(global) The row and column indices in the global matrix <i>X</i> indicating the first row and the first column of the matrix $\text{sub}(X)$, respectively.

<i>descx</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>X</i> .
<i>ipiv</i>	(local) Array of size $LOCr(m_af) + mb_af$. This array contains pivoting information as computed by <code>p?getrf</code> . If $ipiv[i]=j$, then the local row $i+1$ was swapped with the global row j where $i=0, \dots, LOCr(m_af) + mb_af - 1$. This array is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) The array <i>work</i> of size <i>lwork</i> is a workspace array.
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>For real flavors:</i> <i>lwork</i> must be at least $lwork \geq 3 * LOCr(n + \text{mod}(ia-1, mb_a))$ <i>For complex flavors:</i> <i>lwork</i> must be at least $lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a))$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

<i>iwork</i>	(local) Workspace array, size <i>liwork</i> . Used in real flavors only.
<i>liwork</i>	(local or global) The size of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.
<i>rwork</i>	(local) Workspace array, size <i>lrwork</i> . Used in complex flavors only.
<i>lrwork</i>	(local or global) The size of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.

Output Parameters

<i>x</i>	On exit, contains the improved solution vectors.
<i>ferr, berr</i>	Arrays of size $LOCc(jb + nrhs - 1)$ each. The array <i>ferr</i> contains the estimated forward error bound for each solution vector of $\text{sub}(X)$. If <i>XTRUE</i> is the true solution corresponding to $\text{sub}(X)$, <i>ferr</i> is an estimated upper bound for the magnitude of the largest element in $(\text{sub}(X) - XTRUE)$ divided by the magnitude of the largest element in $\text{sub}(X)$. The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.

This array is tied to the distributed matrix X .

The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of $\text{sub}(A)$ or $\text{sub}(B)$ that makes $\text{sub}(X)$ an exact solution). This array is tied to the distributed matrix X .

<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> [0]	On exit, <i>iwork</i> [0] contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> [0]	On exit, <i>rwork</i> [0] contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?porfs

Improves the computed solution to a system of linear equations with symmetric/Hermitian positive definite distributed matrix and provides error bounds and backward error estimates for the solution.

Syntax

```
void psporf (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descaf , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *ferr , float *berr , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pdporf (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descaf , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *ferr , double *berr , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pcporf (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descaf , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *ferr , float *berr , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pzporf (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descaf , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *ferr , double *berr , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?porfs` function improves the computed solution to the system of linear equations

$$\text{sub}(A) * \text{sub}(X) = \text{sub}(B),$$

where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ is a real symmetric or complex Hermitian positive definite distributed matrix and

$$\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+nrhs-1),$$

$$\text{sub}(X) = X(\text{ix}:\text{ix}+n-1, \text{jx}:\text{jx}+nrhs-1)$$

are right-hand side and solution submatrices, respectively. This function also provides error bounds and backward error estimates for the solution.

Input Parameters

<i>uplo</i>	(global) Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored. If <i>uplo</i> = 'U', $\text{sub}(A)$ is upper triangular. If <i>uplo</i> = 'L', $\text{sub}(A)$ is lower triangular.
<i>n</i>	(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) The number of right-hand sides, i.e., the number of columns of the matrices $\text{sub}(B)$ and $\text{sub}(X)$ ($nrhs \geq 0$).
<i>a, af, b, x</i>	(local) Pointers into the local memory to arrays of local sizes $a: lld_a * LOcc(\text{ja}+n-1),$ $af: lld_af * LOcc(\text{jaf}+n-1),$ $b: lld_b * LOcc(\text{jb}+nrhs-1),$ $x: lld_x * LOcc(\text{jx}+nrhs-1).$ The array <i>a</i> contains the local pieces of the <i>n</i> -by- <i>n</i> symmetric/Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced. The array <i>af</i> contains the factors <i>L</i> or <i>U</i> from the Cholesky factorization $\text{sub}(A) = L * L^H$ or $\text{sub}(A) = U^H * U$, as computed by <code>p?potrf</code> . On entry, the array <i>b</i> contains the local pieces of the distributed matrix of right hand sides $\text{sub}(B)$.

On entry, the array *x* contains the local pieces of the solution vectors $\text{sub}(X)$.

ia, ja

(global) The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

desca

(global and local) array of size *dlen_*. The array descriptor for the distributed matrix *A*.

iaf, jaf

(global) The row and column indices in the global matrix *AF* indicating the first row and the first column of the matrix $\text{sub}(AF)$, respectively.

descaf

(global and local) array of size *dlen_*. The array descriptor for the distributed matrix *AF*.

ib, jb

(global) The row and column indices in the global matrix *B* indicating the first row and the first column of the matrix $\text{sub}(B)$, respectively.

descb

(global and local) array of size *dlen_*. The array descriptor for the distributed matrix *B*.

ix, jx

(global) The row and column indices in the global matrix *X* indicating the first row and the first column of the matrix $\text{sub}(X)$, respectively.

descx

(global and local) array of size *dlen_*. The array descriptor for the distributed matrix *X*.

work

(local)

The array *work* of size *lwork* is a workspace array.

lwork

(local) The size of the array *work*.

For real flavors:

lwork must be at least

$lwork \geq 3 * LOCr(n + \text{mod}(ia - 1, mb_a))$

For complex flavors:

lwork must be at least

$lwork \geq 2 * LOCr(n + \text{mod}(ia - 1, mb_a))$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

iwork

(local) Workspace array of size *liwork*. Used in real flavors only.

liwork

(local or global) The size of the array *iwork*; used in real flavors only. Must be at least

$liwork \geq LOCr(n + \text{mod}(ib - 1, mb_b))$.

rwork

(local)

Workspace array of size *lrwork*. Used in complex flavors only.

lrwork

(local or global) The size of the array *rwork*; used in complex flavors only. Must be at least $lrwork \geq LOCr(n + \text{mod}(ib - 1, mb_b))$.

Output Parameters

<i>x</i>	On exit, contains the improved solution vectors.
<i>ferr</i> , <i>berr</i>	<p>Arrays of size $LOC(jb+nrhs-1)$ each.</p> <p>The array <i>ferr</i> contains the estimated forward error bound for each solution vector of sub(X).</p> <p>If XTRUE is the true solution corresponding to sub(X), <i>ferr</i> is an estimated upper bound for the magnitude of the largest element in (sub(X) - XTRUE) divided by the magnitude of the largest element in sub(X). The estimate is as reliable as the estimate for <i>rcond</i>, and is almost always a slight overestimate of the true error.</p> <p>This array is tied to the distributed matrix X.</p> <p>The array <i>berr</i> contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(A) or sub(B) that makes sub(X) an exact solution). This array is tied to the distributed matrix X.</p>
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> [0]	On exit, <i>iwork</i> [0] contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> [0]	On exit, <i>rwork</i> [0] contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	<p>(global) If <i>info</i>=0, the execution is successful.</p> <p><i>info</i> < 0:</p> <p>If the <i>i</i>-th argument is an array and the <i>j</i>-th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = -(<i>j</i>*100+<i>j</i>); if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = -<i>i</i>.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trrfs

Provides error bounds and backward error estimates for the solution to a system of linear equations with a distributed triangular coefficient matrix.

Syntax

```
void pstrrfs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *ferr ,
float *berr , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT
*info );
```

```
void pdtrrfs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
double *ferr , double *berr , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT
*liwork , MKL_INT *info );
```

```

void pctrfs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , float *ferr , float *berr , MKL_Complex8 *work , MKL_INT *lwork ,
float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pztrfs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , double *ferr , double *berr , MKL_Complex16 *work , MKL_INT
*lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );

```

Include Files

- mkl_scalapack.h

Description

The `p?trfs` function provides error bounds and backward error estimates for the solution to one of the systems of linear equations

$$\text{sub}(A) * \text{sub}(X) = \text{sub}(B),$$

$$\text{sub}(A)^T * \text{sub}(X) = \text{sub}(B), \text{ or}$$

$$\text{sub}(A)^H * \text{sub}(X) = \text{sub}(B),$$

where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ is a triangular matrix,

$\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+\text{nrhs}-1)$, and

$\text{sub}(X) = X(\text{ix}:\text{ix}+n-1, \text{jx}:\text{jx}+\text{nrhs}-1)$.

The solution matrix X must be computed by `p?trtrs` or some other means before entering this function. The function `p?trfs` does not do iterative refinement because doing so cannot improve the backward error.

Input Parameters

<i>uplo</i>	(global) Must be 'U' or 'L'. If <i>uplo</i> = 'U', $\text{sub}(A)$ is upper triangular. If <i>uplo</i> = 'L', $\text{sub}(A)$ is lower triangular.
<i>trans</i>	(global) Must be 'N' or 'T' or 'C'. Specifies the form of the system of equations: If <i>trans</i> = 'N', the system has the form $\text{sub}(A) * \text{sub}(X) = \text{sub}(B)$ (No transpose); If <i>trans</i> = 'T', the system has the form $\text{sub}(A)^T * \text{sub}(X) = \text{sub}(B)$ (Transpose); If <i>trans</i> = 'C', the system has the form $\text{sub}(A)^H * \text{sub}(X) = \text{sub}(B)$ (Conjugate transpose).
<i>diag</i>	Must be 'N' or 'U'. If <i>diag</i> = 'N', then $\text{sub}(A)$ is non-unit triangular. If <i>diag</i> = 'U', then $\text{sub}(A)$ is unit triangular.
<i>n</i>	(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).

<i>nrhs</i>	(global) The number of right-hand sides, that is, the number of columns of the matrices $\text{sub}(B)$ and $\text{sub}(X)$ ($nrhs \geq 0$).
<i>a, b, x</i>	<p>(local)</p> <p>Pointers into the local memory to arrays of local sizes</p> <p><i>a</i>: $lld_a * LOCc(ja+n-1)$,</p> <p><i>b</i>: $lld_b * LOCc(jb+nrhs-1)$,</p> <p><i>x</i>: $lld_x * LOCc(jx+nrhs-1)$.</p> <p>The array <i>a</i> contains the local pieces of the original triangular distributed matrix $\text{sub}(A)$.</p> <p>If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.</p> <p>If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.</p> <p>If <i>diag</i> = 'U', the diagonal elements of $\text{sub}(A)$ are also not referenced and are assumed to be 1.</p> <p>On entry, the array <i>b</i> contains the local pieces of the distributed matrix of right hand sides $\text{sub}(B)$.</p> <p>On entry, the array <i>x</i> contains the local pieces of the solution vectors $\text{sub}(X)$.</p>
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>ib, jb</i>	(global) The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the matrix $\text{sub}(B)$, respectively.
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>ix, jx</i>	(global) The row and column indices in the global matrix <i>X</i> indicating the first row and the first column of the matrix $\text{sub}(X)$, respectively.
<i>descx</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>X</i> .
<i>work</i>	<p>(local)</p> <p>The array <i>work</i> of size <i>lwork</i> is a workspace array.</p>
<i>lwork</i>	<p>(local) The size of the array <i>work</i>.</p> <p>For real flavors:</p> <p><i>lwork</i> must be at least $lwork \geq 3 * LOCr(n + \text{mod}(ia-1, mb_a))$</p> <p>For complex flavors:</p> <p><i>lwork</i> must be at least</p>

$$lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a))$$
NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

<i>iwork</i>	(local) Workspace array of size <i>liwork</i> . Used in real flavors only.
<i>liwork</i>	(local or global) The size of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$
<i>rwork</i>	(local) Workspace array of size <i>lrwork</i> . Used in complex flavors only.
<i>lrwork</i>	(local or global) The size of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.

Output Parameters

<i>ferr</i> , <i>berr</i>	Arrays of size $LOCc(jb + nrhs - 1)$ each. The array <i>ferr</i> contains the estimated forward error bound for each solution vector of sub(X). If XTRUE is the true solution corresponding to sub(X), <i>ferr</i> is an estimated upper bound for the magnitude of the largest element in (sub(X) - XTRUE) divided by the magnitude of the largest element in sub(X). The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error. This array is tied to the distributed matrix X. The array <i>berr</i> contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(A) or sub(B) that makes sub(X) an exact solution). This array is tied to the distributed matrix X.
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork</i> [0]	On exit, <i>iwork</i> [0] contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork</i> [0]	On exit, <i>rwork</i> [0] contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) If <i>info</i> =0, the execution is successful. $info < 0$: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then $info = -(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Matrix Inversion: ScaLAPACK Computational Routines

This sections describes ScaLAPACK routines that compute the inverse of a matrix based on the previously obtained factorization. Note that it is not recommended to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see [Solving Systems of Linear Equations](#)); this is more efficient and more accurate.

p?getri

Computes the inverse of a LU-factored distributed matrix.

Syntax

```
void psgetri (MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_INT *ipiv , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *info );

void pdgetri (MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_INT *ipiv , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *info );

void pcgetri (MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT
*liwork , MKL_INT *info );

void pzgetri (MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *liwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?getri function computes the inverse of a general distributed matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ using the LU factorization computed by p?getrf. This method inverts U and then computes the inverse of $\text{sub}(A)$ by solving the system

$$\text{inv}(\text{sub}(A)) * L = \text{inv}(U)$$

for $\text{inv}(\text{sub}(A))$.

Input Parameters

<i>n</i>	(global) The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of local size $lld_a * LOCC(ja+n-1)$. On entry, the array <i>a</i> contains the local pieces of the L and U obtained by the factorization $\text{sub}(A) = P * L * U$ computed by p?getrf.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .

work (local)
The array *work* of size *lwork* is a workspace array.

lwork (local) The size of the array *work*. *lwork* must be at least
 $lwork \geq LOCr(n + \text{mod}(ia-1, mb_a)) * nb_a$.

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

The array *work* is used to keep at most an entire column block of sub(A).

iwork (local) Workspace array used for physically transposing the pivots, size *liwork*.

liwork (local or global) The size of the array *iwork*.

The minimal value *liwork* of is determined by the following code:

```
if NPROW == NPCOL then
    liwork = LOCc(n_a + mod(ja-1, nb_a)) + nb_a
else
    liwork = LOCc(n_a + mod(ja-1, nb_a)) +
    max(ceil(ceil(LOCr(m_a)/mb_a)/(lcm/NPROW)), nb_a)
end if
```

where *lcm* is the least common multiple of process rows and columns (NPROW and NPCOL).

Output Parameters

ipiv (local)
Array of size $LOCr(m_a) + mb_a$.
This array contains the pivoting information.
If *ipiv*[*i*]=*j*, then the local row *i*+1 was swapped with the global row *j* where *i*=0, ... , $LOCr(m_a) + mb_a - 1$.
This array is tied to the distributed matrix A.

work[0] On exit, *work*[0] contains the minimum value of *lwork* required for optimum performance.

iwork[0] On exit, *iwork*[0] contains the minimum value of *liwork* required for optimum performance.

info (global) If *info*=0, the execution is successful.
info < 0:
 If the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:

If $info = i$, the matrix element $U(i,i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?potri

Computes the inverse of a symmetric/Hermitian positive definite distributed matrix.

Syntax

```
void pspotri (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *info );

void pdpotri (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *info );

void pcpotri (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );

void pzpotri (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?potri` function computes the inverse of a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the Cholesky factorization $\text{sub}(A) = U^H * U$ or $\text{sub}(A) = L * L^H$ computed by `p?potrf`.

Input Parameters

<code>uplo</code>	(global) Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored. If <code>uplo = 'U'</code> , upper triangle of $\text{sub}(A)$ is stored. If <code>uplo = 'L'</code> , lower triangle of $\text{sub}(A)$ is stored.
<code>n</code>	(global) The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<code>a</code>	(local) Pointer into the local memory to an array of local size <code>lld_a * LOCC(ja+n-1)</code> . On entry, the array <code>a</code> contains the local pieces of the triangular factor U or L from the Cholesky factorization $\text{sub}(A) = U^H * U$, or $\text{sub}(A) = L * L^H$, as computed by <code>p?potrf</code> .
<code>ia, ja</code>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

desca (global and local) array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

a On exit, overwritten by the local pieces of the upper or lower triangle of the (symmetric/Hermitian) inverse of sub(*A*).

info (global) If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:

If *info* = *i*, the element (*i*, *i*) of the factor *U* or *L* is zero, and the inverse could not be computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trtri

Computes the inverse of a triangular distributed matrix.

Syntax

```
void pstrtri (char *uplo , char *diag , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *info );

void pdtrtri (char *uplo , char *diag , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *info );

void pctrtri (char *uplo , char *diag , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *info );

void pztrtri (char *uplo , char *diag , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?trtri function computes the inverse of a real or complex upper or lower triangular distributed matrix sub(*A*) = *A*(*ia:ia+n-1*, *ja:ja+n-1*).

Input Parameters

uplo (global) Must be 'U' or 'L'.

Specifies whether the distributed matrix sub(*A*) is upper or lower triangular.

If *uplo* = 'U', sub(*A*) is upper triangular.

If *uplo* = 'L', sub(*A*) is lower triangular.

<i>diag</i>	<p>Must be 'N' or 'U'.</p> <p>Specifies whether or not the distributed matrix sub(A) is unit triangular.</p> <p>If <i>diag</i> = 'N', then sub(A) is non-unit triangular.</p> <p>If <i>diag</i> = 'U', then sub(A) is unit triangular.</p>
<i>n</i>	(global) The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(A) ($n \geq 0$).
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of local size $ld_a * LOCc(ja+n-1)$.</p> <p>The array <i>a</i> contains the local pieces of the triangular distributed matrix sub(A).</p> <p>If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of sub(A) contains the upper triangular matrix to be inverted, and the strictly lower triangular part of sub(A) is not referenced.</p> <p>If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of sub(A) contains the lower triangular matrix, and the strictly upper triangular part of sub(A) is not referenced.</p>
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.

Output Parameters

<i>a</i>	On exit, overwritten by the (triangular) inverse of the original matrix.
<i>info</i>	<p>(global) If <i>info</i>=0, the execution is successful.</p> <p><i>info</i> < 0:</p> <p>If the <i>i</i>-th argument is an array and the <i>j</i>-th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = -(<i>i</i>*100+<i>j</i>); if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = -<i>i</i>.</p> <p><i>info</i> > 0:</p> <p>If <i>info</i> = <i>k</i>, the matrix element $A(ia+k-1, ja+k-1)$ is exactly zero. The triangular matrix sub(A) is singular and its inverse cannot be computed.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Matrix Equilibration: ScaLAPACK Computational Routines

ScaLAPACK routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

p?geequ

Computes row and column scaling factors intended to equilibrate a general rectangular distributed matrix and reduce its condition number.

Syntax

```
void psgeequ (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *r , float *c , float *rowcnd , float *colcnd , float *amax , MKL_INT
*info );

void pdgeequ (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *r , double *c , double *rowcnd , double *colcnd , double *amax ,
MKL_INT *info );

void pcgeequ (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *r , float *c , float *rowcnd , float *colcnd , float *amax ,
MKL_INT *info );

void pzgeequ (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *r , double *c , double *rowcnd , double *colcnd , double
*amax , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?geequ` function computes row and column scalings intended to equilibrate an m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ and reduce its condition number. The output array r returns the row scale factors r_i , and the array c returns the column scale factors c_j . These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r_i*a_{ij}*c_j$ have absolute value 1.

r_i and c_j are restricted to be between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of $\text{sub}(A)$ but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the `?lamch` routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

The auxiliary function `p?laqge` uses scaling factors computed by `p?geequ` to scale a general rectangular matrix.

Input Parameters

m	(global) The number of rows to be operated on, that is, the number of rows of the distributed matrix $\text{sub}(A)$ ($m \geq 0$).
n	(global) The number of columns to be operated on, that is, the number of columns of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) Pointer into the local memory to an array of local size $lld_a*LOCc(ja+n-1)$. The array a contains the local pieces of the m -by- n distributed matrix whose equilibration factors are to be computed.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

desca (global and local) array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

r, c (local)
 Arrays of sizes *LOCr(m_a)* and *LOCc(n_a)*, respectively.
 If *info* = 0, or *info* > *ia+m-1*, *r[i]* contain the row scale factors for sub(*A*) for *ia-1* ≤ *i* < *ia+m-1*. *r* is aligned with the distributed matrix *A*, and replicated across every process column. *r* is tied to the distributed matrix *A*.
 If *info* = 0, *c[i]* contain the column scale factors for sub(*A*) for *ja-1* ≤ *i* < *ja+n-1*. *c* is aligned with the distributed matrix *A*, and replicated down every process row. *c* is tied to the distributed matrix *A*.

rowcnd, colcnd (global)
 If *info* = 0 or *info* > *ia+m-1*, *rowcnd* contains the ratio of the smallest *r_i* to the largest *r_i* (*ia* ≤ *i* ≤ *ia+m-1*). If *rowcnd* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *r_i*.
 If *info* = 0, *colcnd* contains the ratio of the smallest *c_j* to the largest *c_j* (*ja* ≤ *j* ≤ *ja+n-1*).
 If *colcnd* ≥ 0.1, it is not worth scaling by *c_j*.

amax (global)
 Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info (global) If *info*=0, the execution is successful.
info < 0:
 If the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
info > 0:
 If *info* = *i* and
i ≤ *m*, the *i*-th row of the distributed matrix
 sub(*A*) is exactly zero;
i > *m*, the (*i* - *m*)-th column of the distributed
 matrix sub(*A*) is exactly zero.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite distributed matrix and reduce its condition number.

Syntax

```
void pspoequ (MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float
*sr , float *sc , float *scond , float *amax , MKL_INT *info );

void pdpoequ (MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
double *sr , double *sc , double *scond , double *amax , MKL_INT *info );

void pcpoequ (MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *sr , float *sc , float *scond , float *amax , MKL_INT *info );

void pzpoequ (MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *sr , double *sc , double *scond , double *amax , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?poequ` function computes row and column scalings intended to equilibrate a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ and reduce its condition number (with respect to the two-norm). The output arrays `sr` and `sc` return the row and column scale factors

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled distributed matrix B with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has ones on the diagonal.

This choice of `sr` and `sc` puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

The auxiliary function `p?laqsy` uses scaling factors computed by `p?geequ` to scale a general rectangular matrix.

Input Parameters

<code>n</code>	(global) The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<code>a</code>	(local) Pointer into the local memory to an array of local size <code>ld_a*LOCc(ja+n-1)</code> . The array <code>a</code> contains the n -by- n symmetric/Hermitian positive definite distributed matrix $\text{sub}(A)$ whose scaling factors are to be computed. Only the diagonal elements of $\text{sub}(A)$ are referenced.
<code>ia, ja</code>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<code>desca</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .

Output Parameters

<code>sr, sc</code>	(local)
---------------------	---------

Arrays of sizes $LOCr(m_a)$ and $LOCc(n_a)$, respectively.

If $info = 0$, the array $sr(ia:ia+n-1)$ contains the row scale factors for $sub(A)$. sr is aligned with the distributed matrix A , and replicated across every process column. sr is tied to the distributed matrix A .

If $info = 0$, the array $sc(ja:ja+n-1)$ contains the column scale factors for $sub(A)$. sc is aligned with the distributed matrix A , and replicated down every process row. sc is tied to the distributed matrix A .

scond

(global)

If $info = 0$, *scond* contains the ratio of the smallest $sr[i]$ (or $sc[j]$) to the largest $sr[i]$ (or $sc[j]$), with

$ia-1 \leq i < ia+n-1$ and $ja-1 \leq j < ja+n-1$.

If $scond \geq 0.1$ and *amax* is neither too large nor too small, it is not worth scaling by sr (or sc).

amax

(global)

Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info

(global)

If $info=0$, the execution is successful.

$info < 0$:

If the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

$info > 0$:

If $info = k$, the k -th diagonal entry of $sub(A)$ is nonpositive.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Orthogonal Factorizations: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for the $QR(RQ)$ and $LQ(QL)$ factorization of matrices. Routines for the RZ factorization as well as for generalized QR and RQ factorizations are also included. For the mathematical definition of the factorizations, see the respective LAPACK sections or refer to [SLUG].

Table "Computational Routines for Orthogonal Factorizations" lists ScaLAPACK routines that perform orthogonal factorization of matrices.

Computational Routines for Orthogonal Factorizations

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	p?geqrf	p?geqpf	p?orgqr p?ungqr	p?ormqr p?unmqr

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, RQ factorization	p?gerqf		p?orgrq p?ungrq	p?ormrq p?unmrq
general matrices, LQ factorization	p?gelqf		p?orglq p?unglq	p?ormlq p?unmlq
general matrices, QL factorization	p?geqlf		p?orgql p?ungql	p?ormql p?unmql
trapezoidal matrices, RZ factorization	p?tzzrf			p?ormrz p?unmrz
pair of matrices, generalized QR factorization	p?ggqrf			
pair of matrices, generalized RQ factorization	p?ggrqf			

p?geqrf

Computes the QR factorization of a general m -by- n matrix.

Syntax

```
void psgeqrf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdgeqrf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

void pcgeqrf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzgeqrf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- mkl_scalapack.h

Description

The p?geqrf function forms the QR factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$A = Q \cdot R$.

Input Parameters

m (global) The number of rows in the distributed matrix $\text{sub}(A)$; ($m \geq 0$).

<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$; ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of local size $\text{lld_a} * \text{LOCc}(j_a + n - 1)$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i>
<i>work</i>	(local). Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $\text{lwork} \geq \text{nb_a} * (\text{mp0} + \text{nq0} + \text{nb_a})$, where $\begin{aligned} \text{iroff} &= \text{mod}(ia-1, \text{mb_a}), \text{icoff} = \text{mod}(ja-1, \text{nb_a}), \\ \text{iarow} &= \text{indxg2p}(ia, \text{mb_a}, \text{MYROW}, \text{rsrc_a}, \text{NPROW}), \\ \text{iacol} &= \text{indxg2p}(ja, \text{nb_a}, \text{MYCOL}, \text{csrc_a}, \text{NPCOL}), \\ \text{mp0} &= \text{numroc}(m + \text{iroff}, \text{mb_a}, \text{MYROW}, \text{iarow}, \text{NPROW}), \\ \text{nq0} &= \text{numroc}(n + \text{icoff}, \text{nb_a}, \text{MYCOL}, \text{iacol}, \text{NPCOL}), \text{ and } \text{numroc}, \\ &\text{indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL} \\ &\text{can be determined by calling the function } \text{blacs_gridinfo}. \end{aligned}$ If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla .

Output Parameters

<i>a</i>	The elements on and above the diagonal of $\text{sub}(A)$ contain the $\min(m, n)$ -by- <i>n</i> upper trapezoidal matrix <i>R</i> (<i>R</i> is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local) Array of size $\text{LOCc}(j_a + \min(m, n) - 1)$. Contains the scalar factor of elementary reflectors. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) = 0, the execution is successful.

< 0 , if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(j_a) * H(j_a+1) * \dots * H(j_a+k-1),$$

where $k = \min(m, n)$.

Each $H(i)$ has the form

$$H(i) = I - \tau u v^* v'$$

where τu is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$, and τu in $\tau u[ja+i-2]$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?geqpf

Computes the QR factorization of a general m -by- n matrix with pivoting.

Syntax

```
void psgeqpf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdgeqpf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

void pcgeqpf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pzgeqpf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?geqpf` function forms the QR factorization with column pivoting of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$$\text{sub}(A) * P = Q * R.$$

Input Parameters

m	(global) The number of rows in the matrix $\text{sub}(A)$ ($m \geq 0$).
n	(global) The number of columns in the matrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) Pointer into the local memory to an array of local size $ld_a * LOCC(ja+n-1)$.

	Contains the local pieces of the distributed matrix sub(A) to be factored.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A(<i>ia:ia+m-1, ja:ja+n-1</i>), respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.
<i>work</i>	(local). Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least

For real flavors:

$$lwork \geq \max(3, mp0 + nq0) + LOCc(ja + n - 1) + nq0.$$

For complex flavors:

$$lwork \geq \max(3, mp0 + nq0).$$

Here

```
iroff = mod(ia-1, mb_a), icoff = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mp0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),
nq0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL),
LOCc(ja+n-1) = numroc(ja+n-1, nb_a, MYCOL, csrc_a, NPCOL),
and numroc, indxg2p are ScaLAPACK tool functions.
```

You can determine MYROW, MYCOL, NPROW and NPCOL by calling the `blacs_gridinfo` function.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

<i>rwork</i>	(local). Workspace array of size <i>lrwork</i> (complex flavors only).
<i>lrwork</i>	(local or global) size of <i>rwork</i> (complex flavors only). The value of <i>lrwork</i> must be at least

$$lwork \geq LOCc(ja + n - 1) + nq0.$$

Here

```
iroff = mod(ia-1, mb_a), icoff = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mp0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),
nq0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL),
```

`LOCc (ja+n-1) = numroc(ja+n-1, nb_a, MYCOL, csrc_a, NPCOL)`, and `numroc`, `indxg2p` are ScaLAPACK tool functions.

You can determine `MYROW`, `MYCOL`, `NPROW` and `NPCOL` by calling the `blacs_gridinfo` function.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<code>a</code>	The elements on and above the diagonal of <code>sub(A)</code> contain the $\min(m,n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array <code>tau</code> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<code>ipiv</code>	(local) Array of size <code>LOCc(ja+n-1)</code> . <code>ipiv[i] = k</code> , the local $(i+1)$ -th column of <code>sub(A)*P</code> was the global k -th column of <code>sub(A)</code> ($0 \leq i < LOCc(ja+n-1)$). <code>ipiv</code> is tied to the distributed matrix A .
<code>tau</code>	(local) Array of size <code>LOCc(ja+min(m, n)-1)</code> . Contains the scalar factor <code>tau</code> of elementary reflectors. <code>tau</code> is tied to the distributed matrix A .
<code>work[0]</code>	On exit, <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>rwork[0]</code>	On exit, <code>rwork[0]</code> contains the minimum value of <code>lrwork</code> required for optimum performance.
<code>info</code>	(global) = 0, the execution is successful. < 0, if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info = -(i*100+j)</code> ; if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(1)*H(2)*...*H(k)$$

where $k = \min(m,n)$.

Each $H(i)$ has the form

$$H = I - \tau * v * v'$$

where `tau` is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in `A(ia+i:ia+m-1, ja+i-1)`.

The matrix P is represented in $ipiv$ as follows: if $ipiv[j] = i$ then the $(j+1)$ -th column of P is the i -th canonical unit vector ($0 \leq j < LOCC(ja+n-1)$).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orgqr

Generates the orthogonal matrix Q of the QR factorization formed by [p?geqrf](#).

Syntax

```
void psorgqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorgqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?orgqr` function generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by [p?geqrf](#).

Input Parameters

m	(global) The number of rows in the matrix $\text{sub}(Q)$ ($m \geq 0$).
n	(global) The number of columns in the matrix $\text{sub}(Q)$ ($m \geq n \geq 0$).
k	(global) The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
a	(local) Pointer into the local memory to an array of local size $ld_a * LOCC(ja+n-1)$. The j -th column of the matrix stored in a must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqrf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
tau	(local) Array of size $LOCC(ja+k-1)$.

Contains the scalar factor $\tau[j]$ of elementary reflectors $H(j+1)$ as returned by `p?geqrf` ($0 \leq j < LOCC(ja+k-1)$). τ is tied to the distributed matrix A .

work

(local)

Workspace array of size of *lwork*.

lwork

(local or global) size of *work*.

Must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where

$iroffa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$,

$iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$,

$mpa0 = \text{numroc}(m+iroffa, mb_a, MYROW, iarow, NPROW)$,

$nqa0 = \text{numroc}(n+icoffa, nb_a, MYCOL, iacol, NPCOL)$;

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

Contains the local pieces of the m -by- n distributed matrix Q .

work[0]

On exit, [0] contains the minimum value of *lwork* required for optimum performance.

info

(global)

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by `p?geqrf`.

Syntax

```
void pcungqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
```

```
void pzungqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```


Include Files

- `mkl_scalapack.h`

Description

This function generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1)*H(2)*...*H(k)$$

as returned by `p?geqrf`.

Input Parameters

<i>m</i>	(global) The number of rows in the matrix sub(Q); ($m \geq 0$).
<i>n</i>	(global) The number of columns in the matrix sub(Q) ($m \geq n \geq 0$).
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size <code>lld_a*LOCc(ja+n-1)</code> . The j -th column of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqrf</code> in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) Array of size <code>LOCc(ja+k-1)</code> . Contains the scalar factor <code>tau[j]</code> of elementary reflectors $H(j+1)$ as returned by <code>p?geqrf</code> ($0 \leq j < LOCc(ja+k-1)$). <i>tau</i> is tied to the distributed matrix A .
<i>work</i>	(local) Workspace array of size of <code>lwork</code> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $lwork \geq nb_a*(nqa0 + mpa0 + nb_a)$, where $iroffa = \text{mod}(ia-1, mb_a),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mpa0 = \text{numroc}(m+iroffa, mb_a, MYROW, iarow, NPROW),$ $nqa0 = \text{numroc}(n+icoffa, nb_a, MYCOL, iacol, NPCOL)$

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q .
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info = -(i*100+j)</code> ; if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormqr

Multiplies a general matrix by the orthogonal matrix Q of the QR factorization formed by `p?geqrf`.

Syntax

```
void psormqr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );

void pdormqr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?ormqr` function overwrites the general real m -by- n distributed matrix sub (C) = $C(ic:ic+m-1,jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * sub(C)$	$sub(C) * Q$
$trans = 'T':$	$Q^T * sub(C)$	$sub(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by [p?geqrf](#). Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	<p>(global)</p> <p>= 'L' : Q or Q^T is applied from the left.</p> <p>= 'R' : Q or Q^T is applied from the right.</p>
<i>trans</i>	<p>(global)</p> <p>= 'N', no transpose, Q is applied.</p> <p>= 'T', transpose, Q^T is applied.</p>
<i>m</i>	(global) The number of rows in the distributed matrix sub(C) ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix sub(C) ($n \geq 0$).
<i>k</i>	<p>(global) The number of elementary reflectors whose product defines the matrix Q. Constraints:</p> <p>If <i>side</i> = 'L', $m \geq k \geq 0$</p> <p>If <i>side</i> = 'R', $n \geq k \geq 0$.</p>
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$. The j-th column of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqrf</code> in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the function but restored on exit.</p> <p>If <i>side</i> = 'L', $lld_a \geq \max(1, LOCr(ia+m-1))$</p> <p>If <i>side</i> = 'R', $lld_a \geq \max(1, LOCr(ia+n-1))$</p>
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>tau</i>	<p>(local)</p> <p>Array of size $LOCc(ja+k-1)$.</p> <p>Contains the scalar factor $\tau[j]$ of elementary reflectors $H(j+1)$ as returned by <code>p?geqrf</code> ($0 \leq j < LOCc(ja+k-1)$). <i>tau</i> is tied to the distributed matrix A.</p>
<i>c</i>	<p>(local)</p> <p>Pointer into the local memory to an array of local size $lld_c * LOCc(jc+n-1)$.</p> <p>Contains the local pieces of the distributed matrix sub(C) to be factored.</p>
<i>ic, jc</i>	(global) The row and column indices in the global matrix C indicating the first row and the first column of the matrix sub(C), respectively.
<i>descc</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix C .
<i>work</i>	(local)

Workspace array of size of *lwork*.

lwork

(local or global) size of *work*, must be at least:

if *side* = 'L',

$lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$

else if *side* = 'R',

$lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(n + icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) * nb_a) + nb_a * nb_a$

end if

where

$lcmq = lcm / NPCOL$ with $lcm = ilcm(NPROW, NPCOL)$,

$iroffa = \text{mod}(ia - 1, mb_a)$,

$icoffa = \text{mod}(ja - 1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$,

$npa0 = \text{numroc}(n + iroffa, mb_a, MYROW, iarow, NPROW)$,

$iroffc = \text{mod}(ic - 1, mb_c)$,

$icoffc = \text{mod}(jc - 1, nb_c)$,

$icrow = \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW)$,

$iccol = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL)$,

$mpc0 = \text{numroc}(m + iroffc, mb_c, MYROW, icrow, NPROW)$,

$nqc0 = \text{numroc}(n + icoffc, nb_c, MYCOL, iccol, NPCOL)$,

$ilcm$, indxg2p and numroc are ScaLAPACK tool functions; $MYROW$, $MYCOL$, $NPROW$ and $NPCOL$ can be determined by calling the function `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p_xerbla`.

Output Parameters

c

Overwritten by the product $Q * \text{sub}(C)$, or $Q^T * \text{sub}(C)$, or $\text{sub}(C) * Q^T$, or $\text{sub}(C) * Q$.

work[0]

On exit *work*[0] contains the minimum value of *lwork* required for optimum performance.

info

(global)

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then *info* = $-(i * 100 + j)$; if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmqr

Multiplies a complex matrix by the unitary matrix Q of the QR factorization formed by `p?geqrf`.

Syntax

```
void pcunmqr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunmqr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

This function overwrites the general complex m -by- n distributed matrix sub (C) = $C(ic:ic+m-1,jc:jc+n-1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
<i>trans</i> = 'T':	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1) H(2) \dots H(k)$ as returned by `p?geqrf`. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) = 'L': Q or Q^H is applied from the left. = 'R': Q or Q^H is applied from the right.
<i>trans</i>	(global) = 'N', no transpose, Q is applied. = 'C', conjugate transpose, Q^H is applied.
<i>m</i>	(global) The number of rows in the distributed matrix sub(C) ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix sub(C) ($n \geq 0$).
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.

<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a * LOCc(ja+k-1)$. The j-th column of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqrf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the function but restored on exit.</p> <p>If <i>side</i> = 'L', $lld_a \geq \max(1, LOCr(ia+m-1))$</p> <p>If <i>side</i> = 'R', $lld_a \geq \max(1, LOCr(ia+n-1))$</p>
<i>ia, ja</i>	<p>(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.</p>
<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p>
<i>tau</i>	<p>(local)</p> <p>Array of size $LOCc(ja+k-1)$.</p> <p>Contains the scalar factor $\tau[j]$ of elementary reflectors $H(j+1)$ as returned by p?geqrf ($0 \leq j < LOCc(ja+k-1)$). <i>tau</i> is tied to the distributed matrix <i>A</i>.</p>
<i>c</i>	<p>(local)</p> <p>Pointer into the local memory to an array of local size $lld_c * LOCc(jc+n-1)$.</p> <p>Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.</p>
<i>ic, jc</i>	<p>(global) The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the submatrix <i>C</i>, respectively.</p>
<i>descc</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>C</i>.</p>
<i>work</i>	<p>(local)</p> <p>Workspace array of size of <i>lwork</i>.</p>
<i>lwork</i>	<p>(local or global) size of <i>work</i>, must be at least:</p> <p>If <i>side</i> = 'L',</p> $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$ <p>else if <i>side</i> = 'R',</p> $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(n + icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) * nb_a) + nb_a * nb_a$ <p>end if</p> <p>where</p> $lcmq = lcm / NPCOL \text{ with } lcm = ilcm(NPROW, NPCOL),$ $iroffa = \text{mod}(ia - 1, mb_a),$ $icoffa = \text{mod}(ja - 1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$

```

npa0 = numroc(n+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL,
NPROW and NPCOL can be determined by calling the function
blacs_gridinfo.

```

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q \cdot \text{sub}(C)$, or $Q^H \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q^H$, or $\text{sub}(C) \cdot Q$.
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i \cdot 100 + j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gelqf

Computes the LQ factorization of a general rectangular matrix.

Syntax

```

void psgelqf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdgelqf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

void pcgelqf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzgelqf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );

```

Include Files

- `mkl_scalapack.h`

Description

The `p?gelqf` function computes the LQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = L*Q$.

Input Parameters

<i>m</i>	(global) The number of rows in the distributed submatrix $\text{sub}(A)$ ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of local size $lld_a * LOCc(ja+n-1)$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) The row and column indices in the global array A indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $lwork \geq mb_a * (mp0 + nq0 + mb_a)$, where $iroff = \text{mod}(ia-1, mb_a),$ $icoff = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mp0 = \text{numroc}(m+iroff, mb_a, MYROW, iarow, NPROW),$ $nq0 = \text{numroc}(n+icoff, nb_a, MYCOL, iacol, NPCOL)$ $\text{indxg2p} \text{ and } \text{numroc} \text{ are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function } \text{blacs_gridinfo}.$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>a</i>	The elements on and below the diagonal of sub(<i>A</i>) contain the <i>m</i> -by- $\min(m,n)$ lower trapezoidal matrix <i>L</i> (<i>L</i> is lower trapezoidal if $m \leq n$); the elements above the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local) Array of size $LOCr(ia+\min(m, n)-1)$. Contains the scalar factors of elementary reflectors. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

$$Q = H(ia+k-1)*H(ia+k-2)*...*H(ia),$$

where $k = \min(m, n)$

Each *H*(*i*) has the form

$$H(i) = I - \tau * v * v'$$

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$, and *tau* in $\tau[ia+i-2]$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
void psorglq (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
             *ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorglq (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
             *ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?orglq` function generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) * \dots * H(2) * H(1)$$

as returned by `p?gelqf`.

Input Parameters

<i>m</i>	(global) The number of rows in the matrix $\text{sub}(Q)$; ($m \geq 0$).
<i>n</i>	(global) The number of columns in the matrix $\text{sub}(Q)$ ($n \geq m \geq 0$).
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of local size $ld_a * LOCc(ja+n-1)$. On entry, the i -th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gelqf</code> in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $lwork \geq mb_a * (mpa0 + nqa0 + mb_a)$, where $iroffa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$, $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$, $mpa0 = \text{numroc}(m + iroffa, mb_a, MYROW, iarow, NPROW)$, $nqa0 = \text{numroc}(n + icoffa, nb_a, MYCOL, iacol, NPCOL)$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a	Contains the local pieces of the m -by- n distributed matrix Q to be factored.
τ	(local) Array of size $LOCr(ia+k-1)$. Contains the scalar factors $\tau[j]$ of elementary reflectors $H(j+1)$, $0 \leq j < LOCr(ia+k-1)$. τ is tied to the distributed matrix A .
$work[0]$	On exit, $work[0]$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unglq

Generates the unitary matrix Q of the LQ factorization formed by [p?gelqf](#).

Syntax

```
void pcunglq (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
```

```
void pzunglq (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

This function generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1,ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = (H(k))^H \dots (H(2))^H (H(1))^H$ as returned by [p?gelqf](#).

Input Parameters

m	(global) The number of rows in the matrix $\text{sub}(Q)$ ($m \geq 0$).
-----	---

<i>n</i>	(global) The number of columns in the matrix $\text{sub}(Q)$ ($n \geq m \geq 0$).
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of local size $ld_a * LOCC(ja+n-1)$. On entry, the <i>i</i> -th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) Array of size $LOCr(ia+k-1)$. Contains the scalar factors $\tau[j]$ of elementary reflectors $H(j+1)$, $0 \leq j < LOCr(ia+k-1)$. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $lwork \geq mb_a * (mpa0 + nqa0 + mb_a)$, where $iroffa = \text{mod}(ia-1, mb_a),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mpa0 = \text{numroc}(m + iroffa, mb_a, MYROW, iarow, NPROW),$ $nqa0 = \text{numroc}(n + icoffa, nb_a, MYCOL, iacol, NPCOL)$ indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function <code>blacs_gridinfo</code> .

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<i>a</i>	Contains the local pieces of the m -by- n distributed matrix Q to be factored.
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormlq

Multiplies a general matrix by the orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
void psormlq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );

void pdormlq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?ormlq function overwrites the general real m -by- n distributed matrix $sub(C) = C(ic:ic+m-1,jc:jc+n-1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q * sub(C)$	$sub(C) * Q$
<i>trans</i> = 'T':	$Q^T * sub(C)$	$sub(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(k) \dots H(2) H(1)$$

as returned by p?gelqf. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) = 'L': Q or Q^T is applied from the left. = 'R': Q or Q^T is applied from the right.
<i>trans</i>	(global)

	='N', no transpose, Q is applied.
	='T', transpose, Q^T is applied.
<i>m</i>	(global) The number of rows in the distributed matrix sub(C) ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix sub(C) ($n \geq 0$).
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja+m-1)$, if <i>side</i> = 'L' and $lld_a * LOCC(ja+n-1)$, if <i>side</i> = 'R'. The i -th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) Array of size $LOCC(ja+k-1)$. Contains the scalar factor $\tau[j]$ of elementary reflectors $H(j+1)$ as returned by p?gelqf ($0 \leq j < LOCC(ja+k-1)$). <i>tau</i> is tied to the distributed matrix A .
<i>c</i>	(local) Pointer into the local memory to an array of local size $lld_c * LOCC(jc+n-1)$. Contains the local pieces of the distributed matrix sub(C) to be factored.
<i>ic, jc</i>	(global) The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C , respectively.
<i>descc</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix C .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) size of the array <i>work</i> ; must be at least: If <i>side</i> = 'L', $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + maxmqa0) + \text{numroc}(\text{numroc}(m + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0)) * mb_a) + mb_a * mb_a$ else if <i>side</i> = 'R',

```

lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0+nqc0)*mb_a + mb_a*mb_a)
end if
where
lcm = lcm/NPROW with lcm = ilcm (NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(m+icoffa, nb_a, MYCOL, iacol, NPCOL),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork` = -1, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q \cdot \text{sub}(C)$, or $Q' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q'$, or $\text{sub}(C) \cdot Q$
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i \cdot 100 + j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmlq

Multiplies a general matrix by the unitary matrix Q of the LQ factorization formed by p?gelqf.

Syntax

```
void pcunmlq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunmlq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

This function overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by p?gelqf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) = $'L'$: Q or Q^H is applied from the left. = $'R'$: Q or Q^H is applied from the right.
$trans$	(global) = $'N'$, no transpose, Q is applied. = $'C'$, conjugate transpose, Q^H is applied.
m	(global) The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
n	(global) The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
k	(global) The number of elementary reflectors whose product defines the matrix Q . Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.
a	(local)

Pointer into the local memory to an array of size $lld_a * LOCc(ja+m-1)$, if $side = 'L'$ and $lld_a * LOCc(ja+n-1)$, if $side = 'R'$, where $lld_a \geq \max(1, LOCr(ia+k-1))$. The i -th column of the matrix stored in a must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by `p?gelqf` in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.

<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	(local) Array of size $LOCc(ia+k-1)$. Contains the scalar factor $tau[j]$ of elementary reflectors $H(j+1)$ as returned by <code>p?gelqf</code> ($0 \leq j < LOCc(ia+k-1)$). tau is tied to the distributed matrix A .
<i>c</i>	(local) Pointer into the local memory to an array of local size $lld_c * LOCc(jc+n-1)$. Contains the local pieces of the distributed matrix sub(C) to be factored.
<i>ic, jc</i>	(global) The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C , respectively.
<i>descc</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix C .
<i>work</i>	(local) Workspace array of size of $lwork$.
<i>lwork</i>	(local or global) size of the array <i>work</i> ; must be at least: If $side = 'L'$, $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + maxmq0) + \text{numroc}(\text{numroc}(m + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcm), nqc0) * mb_a + mb_a * mb_a$ else if $side = 'R'$, $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a + mb_a * mb_a$ end if where $lcmp = lcm / NPROW \text{ with } lcm = ilcm(NPROW, NPCOL),$ $iroffa = \text{mod}(ia - 1, mb_a),$ $icoffa = \text{mod}(ja - 1, nb_a),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mq0 = \text{numroc}(m + icoffa, nb_a, MYCOL, iacol, NPCOL),$ $iroffc = \text{mod}(ic - 1, mb_c),$

```

icoffc = mod(jc-1, nb_c),
icrow = indxcg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxcg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+icoffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

`mod(x,y)` is the integer remainder of x/y .

`ilcm`, `indxcg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q \cdot \text{sub}(C)$, or $Q' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q'$, or $\text{sub}(C) \cdot Q$
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i \cdot 100 + j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?geqlf

Computes the QL factorization of a general matrix.

Syntax

```

void psgeqlf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdgeqlf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

void pcgeqlf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzgeqlf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );

```

Include Files

- `mkl_scalapack.h`

Description

The `p?geqlf` function forms the QL factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1) = Q^*L$.

Input Parameters

<code>m</code>	(global) The number of rows in the matrix $\text{sub}(Q)$; ($m \geq 0$).
<code>n</code>	(global) The number of columns in the matrix $\text{sub}(Q)$ ($n \geq 0$).
<code>a</code>	(local) Pointer into the local memory to an array of local size $ld_a * LOCc(\text{ja}+n-1)$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<code>ia, ja</code>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1)$, respectively.
<code>desca</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .
<code>work</code>	(local) Workspace array of size of <code>lwork</code> .
<code>lwork</code>	(local or global) size of <code>work</code> , must be at least $lwork \geq nb_a * (mp0 + nq0 + nb_a)$, where $iroff = \text{mod}(\text{ia}-1, mb_a),$ $icoff = \text{mod}(\text{ja}-1, nb_a),$ $iarow = \text{indxg2p}(\text{ia}, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}),$ $iacol = \text{indxg2p}(\text{ja}, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}),$ $mp0 = \text{numroc}(m+iroff, mb_a, \text{MYROW}, iarow, \text{NPROW}),$ $nq0 = \text{numroc}(n+icoff, nb_a, \text{MYCOL}, iacol, \text{NPCOL})$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`numroc` and `indxg2p` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<i>a</i>	On exit, if $m \geq n$, the lower triangle of the distributed submatrix $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the n -by- n lower triangular matrix L ; if $m \leq n$, the elements on and below the $(n-m)$ -th superdiagonal contain the m -by- n lower trapezoidal matrix L ; the remaining elements, with the array <i>tau</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local) Array of size $LOCc(ja+n-1)$. Contains the scalar factors of elementary reflectors. <i>tau</i> is tied to the distributed matrix A .
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja+k-1) * \dots * H(ja+1) * H(ja)$$

where $k = \min(m, n)$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(m-k+i+1:m) = 0$ and $v(m-k+i) = 1$; $v(1:m-k+i-1)$ is stored on exit in $A(ia:ia+m-k+i-2, ja+n-k+i-1)$, and τ in $\tau[ja+n-k+i-2]$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orgql

Generates the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
void psorgql (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorgql (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?orgql` function generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) * \dots * H(2) * H(1)$$

as returned by `p?geqlf`.

Input Parameters

<i>m</i>	(global) The number of rows in the matrix $\text{sub}(Q)$, ($m \geq 0$).
<i>n</i>	(global) The number of columns in the matrix $\text{sub}(Q)$, ($m \geq n \geq 0$).
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of local size $ld_a * LOCc(ja+n-1)$. On entry, the j -th column of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(j), ja+n-k \leq j \leq ja+n-1$, as returned by <code>p?geqlf</code> in the k columns of its distributed matrix argument $A(ia:*, ja+n-k:ja+n-1)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) Array of size $LOCc(ja+n-1)$. Contains the scalar factors $\tau[j]$ of elementary reflectors $H(j+1)$, $0 \leq j < LOCr(ia+n-1)$. <i>tau</i> is tied to the distributed matrix A .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where $iroffa = \text{mod}(ia-1, mb_a),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mpa0 = \text{numroc}(m + iroffa, mb_a, MYROW, iarow, NPROW),$ $nqa0 = \text{numroc}(n + icoffa, nb_a, MYCOL, iacol, NPCOL)$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q to be factored.
<code>work[0]</code>	On exit, <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info = -(i*100+j)</code> ; if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ungql

Generates the unitary matrix Q of the QL factorization formed by `p?geqlf`.

Syntax

```
void pcungql (const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , MKL_Complex8
*a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8
*tau , MKL_Complex8 *work , const MKL_INT *lwork , MKL_INT *info );

void pzungql (const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , MKL_Complex16
*a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16
*tau , MKL_Complex16 *work , const MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkc_scalapack.h`

Description

This function generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first n columns of a product of k elementary reflectors of order m

$Q = (H(k))^H \dots (H(2))^H (H(1))^H$ as returned by `p?geqlf`.

Input Parameters

<i>m</i>	(global) The number of rows in the matrix sub(<i>Q</i>) ($m \geq 0$).
<i>n</i>	(global) The number of columns in the matrix sub(<i>Q</i>) ($m \geq n \geq 0$).
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix <i>Q</i> ($n \geq k \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of local size <i>ld_a</i> * <i>LOCc</i> (<i>ja</i> + <i>n</i> - 1). On entry, the <i>j</i> -th column of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector <i>H</i> (<i>j</i>), $ja + n - k \leq j \leq ja + n - 1$, as returned by <code>p?geqlf</code> in the <i>k</i> columns of its distributed matrix argument <i>A</i> (<i>ia</i> : *, <i>ja</i> + <i>n</i> - <i>k</i> : <i>ja</i> + <i>n</i> - 1).
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> (<i>ia</i> : <i>ia</i> + <i>m</i> - 1, <i>ja</i> : <i>ja</i> + <i>n</i> - 1), respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) Array of size <i>LOCr</i> (<i>ia</i> + <i>n</i> - 1). Contains the scalar factors <i>tau</i> [<i>j</i>] of elementary reflectors <i>H</i> (<i>j</i> + 1), $0 \leq j < \text{LOCr}(ia + n - 1)$. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where $iroffa = \text{mod}(ia - 1, mb_a),$ $icoffa = \text{mod}(ja - 1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}),$ $iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}),$ $mpa0 = \text{numroc}(m + iroffa, mb_a, \text{MYROW}, iarow, \text{NPROW}),$ $nqa0 = \text{numroc}(n + icoffa, nb_a, \text{MYCOL}, iacol, \text{NPCOL})$ <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function <code>blacs_gridinfo</code> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pxerbla</code> .

Output Parameters

<i>a</i>	Contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix <i>Q</i> to be factored.
----------	---

`work[0]` On exit, `work[0]` contains the minimum value of `lwork` required for optimum performance.

`info` (global)
 = 0: the execution is successful.
 < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then `info` = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then `info` = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormql

Multiplies a general matrix by the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
void psormql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );

void pdormql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?ormql` function overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by `p?geqlf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

`side` (global)
 = 'L': Q or Q^T is applied from the left.
 = 'R': Q or Q^T is applied from the right.

`trans` (global)
 = 'N', no transpose, Q is applied.
 = 'T', transpose, Q^T is applied.

<i>m</i>	(global) The number of rows in the distributed matrix sub(<i>C</i>), ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix sub(<i>C</i>), ($n \geq 0$).
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix <i>Q</i> . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja+k-1)$. The <i>j</i> -th column of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqlf</code> in the <i>k</i> columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the function but restored on exit. If <i>side</i> = 'L', $lld_a \geq \max(1, LOCr(ia+m-1))$, If <i>side</i> = 'R', $lld_a \geq \max(1, LOCr(ia+n-1))$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) Array of size $LOCC(ja+n-1)$. Contains the scalar factor $\tau[j]$ of elementary reflectors $H(j+1)$ as returned by <code>p?geqlf</code> ($0 \leq j < LOCC(ja+k-1)$). <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) Pointer into the local memory to an array of local size $lld_c * LOCC(jc+n-1)$. Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.
<i>ic, jc</i>	(global) The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descc</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) dimension of <i>work</i> , must be at least: If <i>side</i> = 'L', $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a + nb_a * nb_a)$ else if <i>side</i> = 'R', $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(n + icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) * nb_a + nb_a * nb_a)$

```

end if
where
lcmq = lcm/NPCOL with lcm = ilcm (NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
npa0= numroc(n + iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

`mod(x,y)` is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork` = -1, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p?erbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q'^* \text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmql

Multiplies a general matrix by the unitary matrix Q of the QL factorization formed by p?geqlf.

Syntax

```
void pcunmql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunmql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

This function overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'C':$	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by `p?geqlf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) = $'L'$: Q or Q^H is applied from the left. = $'R'$: Q or Q^H is applied from the right.
$trans$	(global) = $'N'$, no transpose, Q is applied. = $'C'$, conjugate transpose, Q^H is applied.
m	(global) The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
n	(global) The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
k	(global) The number of elementary reflectors whose product defines the matrix Q . Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.
a	(local)

Pointer into the local memory to an array of size $lld_a * LOCc(ja+k-1)$. The j -th column of the matrix stored in a must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by `p?geqlf` in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the function but restored on exit.

If $side = 'L'$, $lld_a \geq \max(1, LOCr(ia+m-1))$,

If $side = 'R'$, $lld_a \geq \max(1, LOCr(ia+n-1))$.

ia, ja

(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

desca

(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

tau

(local)

Array of size $LOCc(ia+n-1)$.

Contains the scalar factor $tau[j]$ of elementary reflectors $H(j+1)$ as returned by `p?geqlf` ($0 \leq j < LOCc(ia+n-1)$). tau is tied to the distributed matrix A .

c

(local)

Pointer into the local memory to an array of local size $lld_c * LOCc(jc+n-1)$.

Contains the local pieces of the distributed matrix $sub(C)$ to be factored.

ic, jc

(global) The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C , respectively.

descc

(global and local) array of size $dlen_$. The array descriptor for the distributed matrix C .

work

(local)

Workspace array of size of $lwork$.

lwork

(local or global) size of $work$, must be at least:

If $side = 'L'$,

$lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a + nb_a * nb_a)$

else if $side = 'R'$,

$lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + maxnpa0) + \text{numroc}(\text{numroc}(n + icoffa, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmpq), mpc0)) * nb_a + nb_a * nb_a$

end if

where

$lcmp = lcm / NPCOL$ with $lcm = ilcm(NPROW, NPCOL)$,

$iroffa = \text{mod}(ia - 1, mb_a)$,

$icoffa = \text{mod}(ja - 1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$,

$npa0 = \text{numroc}(n + iroffa, mb_a, MYROW, iarow, NPROW)$,

```

iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

`mod(x,y)` is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

NOTE

`mod(x,y)` is the integer remainder of x/y .

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q' \text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gerqf

Computes the RQ factorization of a general rectangular matrix.

Syntax

```

void psgerqf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdgerqf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

```

```
void pcgerqf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzgerqf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- mkl_scalapack.h

Description

The `p?gerqf` function forms the *QR* factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$$A = R * Q$$

Input Parameters

<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$; ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$; ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of local size $lld_a * LOCC(ja+n-1)$. Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i>
<i>work</i>	(local). Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $lwork \geq mb_a * (mp0 + nq0 + mb_a)$, where $iroff = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW)$, $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL)$, $mp0 = \text{numroc}(m + iroff, mb_a, MYROW, iarow, NPROW)$,

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

$nq0 = \text{numroc}(n + icoff, nb_a, MYCOL, iacol, NPCOL)$ and numroc , indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<code>a</code>	On exit, if $m \leq n$, the upper triangle of $A(ia:ia+m-1, ja:ja+n-1)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m - n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array <code>tau</code> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<code>tau</code>	(local) Array of size <code>LOCr(ia+m-1)</code> . Contains the scalar factor of elementary reflectors. <code>tau</code> is tied to the distributed matrix A .
<code>work[0]</code>	On exit, <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0, the execution is successful. < 0, if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ia) * H(ia+1) * \dots * H(ia+k-1),$$

where $k = \min(m, n)$.

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v'$$

where `tau` is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and `tau` in `tau[ia+m-k+i-2]`.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orgqr

Generates the orthogonal matrix Q of the RQ factorization formed by [p?gerqf](#).

Syntax

```
void psorgqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorgqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?orgqrqf` function generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows that is defined as the last m rows of a product of k elementary reflectors of order n

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by `p?gerqf`.

Input Parameters

<i>m</i>	(global) The number of rows in the matrix $\text{sub}(Q)$, ($m \geq 0$).
<i>n</i>	(global) The number of columns in the matrix $\text{sub}(Q)$, ($n \geq m \geq 0$).
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of local size $ld_a * LOCC(ja+n-1)$. The i -th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+m-1$, as returned by <code>p?gerqf</code> in the k rows of its distributed matrix argument $A(ia+m-k:ia+m-1, ja:*)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) Array of size $LOCC(ja+k-1)$. Contains the scalar factor $\tau[i]$ of elementary reflectors $H(i+1)$ as returned by <code>p?gerqf</code> , $0 \leq i < LOCr(ja+k-1)$. <i>tau</i> is tied to the distributed matrix A .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $lwork \geq mb_a * (mpa0 + nqa0 + mb_a)$, where $iroffa = \text{mod}(ia-1, mb_a),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mpa0 = \text{numroc}(m+iroffa, mb_a, MYROW, iarow, NPROW),$ $nqa0 = \text{numroc}(n+icoffa, nb_a, MYCOL, iacol, NPCOL)$

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

NOTE

`mod(x, y)` is the integer remainder of x/y .

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q .
<code>work[0]</code>	On exit, <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info = -(i*100+j)</code> ; if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ungrq

Generates the unitary matrix Q of the RQ factorization formed by `p?gerqf`.

Syntax

```
void pcungrq (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
```

```
void pzungrq (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

This function generates the m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ as returned by [p?gerqf](#).

Input Parameters

<code>m</code>	(global) The number of rows in the matrix <code>sub(Q)</code> ; ($m \geq 0$).
----------------	---

<i>n</i>	(global) The number of columns in the matrix $\text{sub}(Q)$ ($n \geq m \geq 0$).
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size $\text{ld}_a * \text{LOCc}(j_a + n - 1)$. The i -th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia + m - k \leq i \leq ia + m - 1$, as returned by p?gerqf in the k rows of its distributed matrix argument $A(ia + m - k : ia + m - 1, ja : *)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size dlen_- . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) Array of size $\text{LOCr}(ia + m - 1)$. Contains the scalar factor $\text{tau}[i]$ of elementary reflectors $H(i+1)$ as returned by p?gerqf , $0 \leq i < \text{LOCr}(ia + m - 1)$. <i>tau</i> is tied to the distributed matrix A .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $\text{lwork} \geq \text{mb}_a * (\text{mpa0} + \text{nqa0} + \text{mb}_a)$, where $\text{iroffa} = \text{mod}(ia - 1, \text{mb}_a),$ $\text{icoffa} = \text{mod}(ja - 1, \text{nb}_a),$ $\text{iarow} = \text{indxg2p}(ia, \text{mb}_a, \text{MYROW}, \text{rsrc}_a, \text{NPROW}),$ $\text{iacol} = \text{indxg2p}(ja, \text{nb}_a, \text{MYCOL}, \text{csrc}_a, \text{NPCOL}),$ $\text{mpa0} = \text{numroc}(m + \text{iroffa}, \text{mb}_a, \text{MYROW}, \text{iarow}, \text{NPROW}),$ $\text{nqa0} = \text{numroc}(n + \text{icoffa}, \text{nb}_a, \text{MYCOL}, \text{iacol}, \text{NPCOL})$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

[indxg2p](#) and [numroc](#) are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function [blacs_gridinfo](#).

If $\text{lwork} = -1$, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>a</i>	Contains the local pieces of the m -by- n distributed matrix Q .
----------	--

`work[0]` On exit `work[0]` contains the minimum value of `lwork` required for optimum performance.

`info` (global)

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then `info` = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then `info` = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormr3

Applies an orthogonal distributed matrix to a general m -by- n distributed matrix.

Syntax

```
void psormr3 (const char* side, const char* trans, const MKL_INT* m, const MKL_INT* n,
const MKL_INT* k, const MKL_INT* l, const float* a, const MKL_INT* ia, const MKL_INT*
ja, const MKL_INT* desca, const float* tau, float* c, const MKL_INT* ic, const MKL_INT*
jc, const MKL_INT* descc, float* work, const MKL_INT* lwork, MKL_INT* info);

void pdormr3 (const char* side, const char* trans, const MKL_INT* m, const MKL_INT* n,
const MKL_INT* k, const MKL_INT* l, const double* a, const MKL_INT* ia, const MKL_INT*
ja, const MKL_INT* desca, const double* tau, double* c, const MKL_INT* ic, const
MKL_INT* jc, const MKL_INT* descc, double* work, const MKL_INT* lwork, MKL_INT* info);
```

Include Files

- `mkl_scalapack.h`

Description

`p?ormr3` overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N'$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T'$	$Q^T * \text{sub}(C)$ $Q * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by `p?tzzrf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

`side` (global)

= 'L': apply Q or Q^T from the Left;

= 'R': apply Q or Q^T from the Right.

<i>trans</i>	<p>(global)</p> <p>= 'N': No transpose, apply Q;</p> <p>= 'T': Transpose, apply Q^T.</p>
<i>m</i>	<p>(global)</p> <p>The number of rows to be operated on i.e the number of rows of the distributed submatrix sub(C). $m \geq 0$.</p>
<i>n</i>	<p>(global)</p> <p>The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(C). $n \geq 0$.</p>
<i>k</i>	<p>(global)</p> <p>The number of elementary reflectors whose product defines the matrix Q.</p> <p>If $side = 'L', m \geq k \geq 0$,</p> <p>if $side = 'R', n \geq k \geq 0$.</p>
<i>l</i>	<p>(global)</p> <p>The columns of the distributed submatrix sub(A) containing the meaningful part of the Householder reflectors.</p> <p>If $side = 'L', m \geq l \geq 0$,</p> <p>if $side = 'R', n \geq l \geq 0$.</p>
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a * LOCc(ja+m-1)$ if $side='L'$, and $lld_a * LOCc(ja+n-1)$ if $side='R'$, where $lld_a \geq \text{MAX}(1, \text{LOCr}(ia+k-1))$;</p> <p>On entry, the i-th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?tzrzf</code> in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.</p> <p>$A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.</p>
<i>ia</i>	<p>(global)</p> <p>The row index in the global array a indicating the first row of sub(A).</p>
<i>ja</i>	<p>(global)</p> <p>The column index in the global array a indicating the first column of sub(A).</p>
<i>desca</i>	<p>(global and local)</p> <p>Array of size $dlen_$.</p> <p>The array descriptor for the distributed matrix A.</p>
<i>tau</i>	<p>(local)</p> <p>Array, size $\text{LOCc}(ia+k-1)$.</p> <p>This array contains the scalar factors $tau(i)$ of the elementary reflectors $H(i)$ as returned by <code>p?tzrzf</code>. tau is tied to the distributed matrix A.</p>

<i>c</i>	(local) Pointer into the local memory to an array of size $ld_c * LOCc(jc+n-1)$. On entry, the local pieces of the distributed matrix sub(<i>C</i>).
<i>ic</i>	(global) The row index in the global array <i>c</i> indicating the first row of sub(<i>C</i>).
<i>jc</i>	(global) The column index in the global array <i>c</i> indicating the first column of sub(<i>C</i>).
<i>desc</i>	(global and local) Array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) Array, size (<i>lwork</i>)
<i>lwork</i>	(local) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least If <i>side</i> = 'L', $lwork \geq MpC0 + \text{MAX}(\text{MAX}(1, NqC0), \text{numroc}(\text{numroc}(m + IROFFC, mb_a, 0, 0, NPROW), mb_a, 0, 0, NqC0))$; if <i>side</i> = 'R', $lwork \geq NqC0 + \text{MAX}(1, MpC0)$; where $LCMP = LCM / NPROW$ $LCM = \text{iclcm}(NPROW, NPCOL)$, $IROFFC = \text{MOD}(ic-1, mb_c)$, $ICOFFC = \text{MOD}(jc-1, nb_c)$, $ICROW = \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW)$, $ICCOL = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL)$, $MpC0 = \text{numroc}(m + IROFFC, mb_c, MYROW, ICROW, NPROW)$, $NqC0 = \text{numroc}(n + ICOFFC, nb_c, MYCOL, ICCOL, NPCOL)$, <i>iclcm</i> , <i>indxg2p</i> , and <i>numroc</i> are ScaLAPACK tool functions; <i>MYROW</i> , <i>MYCOL</i> , <i>NPROW</i> and <i>NPCOL</i> can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>pxerbla</i> .

Output Parameters

<i>c</i>	On exit, sub(<i>C</i>) is overwritten by $Q * \text{sub}(C)$ or $Q' * \text{sub}(C)$ or $\text{sub}(C) * Q'$ or $\text{sub}(C) * Q$.
----------	---

work On exit, *work*[0] returns the minimal and optimal *lwork*.

info (local)

= 0: successful exit

< 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = $-(i*100+j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

Application Notes

Alignment requirements

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L',

(*nb_a* = *mb_c* .AND. ICOFFA = IROFFC)

If *side* = 'R',

(*nb_a* = *nb_c* .AND. ICOFFA = ICOFFC .AND. IACOL = ICCOL)

p?unmr3

Applies an orthogonal distributed matrix to a general m-by-n distributed matrix.

Syntax

```
void pcunmr3 (const char* side, const char* trans, const MKL_INT* m, const MKL_INT* n,
const MKL_INT* k, const MKL_INT* l, const MKL_Complex8* a, const MKL_INT* ia, const
MKL_INT* ja, const MKL_INT* desca, const MKL_Complex8* tau, MKL_Complex8* c, const
MKL_INT* ic, const MKL_INT* jc, const MKL_INT* descc, MKL_Complex8* work, const
MKL_INT* lwork, MKL_INT* info);
```

```
void pzunmr3 (const char* side, const char* trans, const MKL_INT* m, const MKL_INT* n,
const MKL_INT* k, const MKL_INT* l, const MKL_Complex16* a, const MKL_INT* ia, const
MKL_INT* ja, const MKL_INT* desca, const MKL_Complex16* tau, MKL_Complex16* c, const
MKL_INT* ic, const MKL_INT* jc, const MKL_INT* descc, MKL_Complex16* work, const
MKL_INT* lwork, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

p?unmr3 overwrites the general complex *m*-by-*n* distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

side = 'L' *side* = 'R'

trans = 'N': $Q * \text{sub}(C)$ $\text{sub}(C) * Q$

trans = 'C': $Q^H * \text{sub}(C)$ $\text{sub}(C) * Q^H$

where *Q* is a complex unitary distributed matrix defined as the product of *k* elementary reflectors

$Q = H(1)' H(2)' \dots H(k)'$

as returned by p?tzzrf. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

<i>side</i>	<p>(global)</p> <p>= 'L': apply Q or Q^H from the Left;</p> <p>= 'R': apply Q or Q^H from the Right.</p>
<i>trans</i>	<p>(global)</p> <p>= 'N': No transpose, apply Q;</p> <p>= 'C': Conjugate transpose, apply Q^H.</p>
<i>m</i>	<p>(global)</p> <p>The number of rows to be operated on i.e the number of rows of the distributed submatrix $sub(C)$. $m \geq 0$.</p>
<i>n</i>	<p>(global)</p> <p>The number of columns to be operated on i.e the number of columns of the distributed submatrix $sub(C)$. $n \geq 0$.</p>
<i>k</i>	<p>(global)</p> <p>The number of elementary reflectors whose product defines the matrix Q.</p> <p>If $side = 'L', m \geq k \geq 0$, if $side = 'R', n \geq k \geq 0$.</p>
<i>l</i>	<p>(global)</p> <p>The columns of the distributed submatrix $sub(A)$ containing the meaningful part of the Householder reflectors.</p> <p>If $side = 'L', m \geq l \geq 0$, if $side = 'R', n \geq l \geq 0$.</p>
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a * LOCC(ja+m-1)$ if $side='L'$, and $lld_a * LOCC(ja+n-1)$ if $side='R'$, where $lld_a \geq \text{MAX}(1, \text{LOCr}(ia+k-1))$;</p> <p>On entry, the i-th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?tzrzf</code> in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.</p> <p>$A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.</p>
<i>ia</i>	<p>(global)</p> <p>The row index in the global array <i>a</i> indicating the first row of $sub(A)$.</p>
<i>ja</i>	<p>(global)</p> <p>The column index in the global array <i>a</i> indicating the first column of $sub(A)$.</p>
<i>desca</i>	<p>(global and local)</p> <p>Array of size <i>dlen_</i>.</p> <p>The array descriptor for the distributed matrix <i>A</i>.</p>
<i>tau</i>	<p>(local)</p> <p>Array, size $\text{LOCc}(ia+k-1)$.</p>

This array contains the scalar factors $\tau(i)$ of the elementary reflectors $H(i)$ as returned by `p?tzrzf`. τ is tied to the distributed matrix A .

c

(local)

Pointer into the local memory to an array of size $lld_c * LOCc(jc+n-1)$.

On entry, the local pieces of the distributed matrix $\text{sub}(C)$.

ic

(global)

The row index in the global array *c* indicating the first row of $\text{sub}(C)$.

jc

(global)

The column index in the global array *c* indicating the first column of $\text{sub}(C)$.

desc

(global and local)

Array of size *dlen*.

The array descriptor for the distributed matrix C .

work

(local)

Array, size (*lwork*)

On exit, *work*(1) returns the minimal and optimal *lwork*.

lwork

(local or global)

The size of the array *work*.

lwork is local input and must be at least

If *side* = 'L', $lwork \geq MpC0 + \text{MAX}(\text{MAX}(1, NqC0), \text{numroc}(\text{numroc}(m + IROFFC, mb_a, 0, 0, NPROW), mb_a, 0, 0, LCMP))$;

if *side* = 'R', $lwork \geq NqC0 + \text{MAX}(1, MpC0)$;

where $LCMP = LCM / NPROW$ with $LCM = \text{ICLM}(NPROW, NPCOL)$,

$IROFFC = \text{MOD}(ic-1, MB_C)$, $ICOFFC = \text{MOD}(jc-1, nb_c)$,

$ICROW = \text{indxg2p}(ic, MB_C, MYROW, rsrc_c, NPROW)$,

$ICCOL = \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL)$,

$MpC0 = \text{numroc}(m + IROFFC, MB_C, MYROW, ICROW, NPROW)$,

$NqC0 = \text{numroc}(n + ICOFFC, nb_c, MYCOL, ICCOL, NPCOL)$,

ilcm, *indxg2p*, and *numroc* are ScaLAPACK tool functions;

MYROW, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `p?xerbla`.

Output Parameters

<i>c</i>	On exit, <code>sub(C)</code> is overwritten by $Q * \text{sub}(C)$ or $Q' * \text{sub}(C)$ or $\text{sub}(C) * Q'$ or $\text{sub}(C) * Q$.
<i>work</i>	(local) Array, size (<i>lwork</i>) On exit, <code>work[0]</code> returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) = 0: successful exit < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <code>info</code> = $-(i * 100 + j)$, if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

Application Notes

Alignment requirements

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (*nb_a* = MB_C and ICOFFA = IROFFC)

If *side* = 'R', (*nb_a* = *nb_c* and ICOFFA = ICOFFC and IACOL = ICCOL)

p?ormrq

Multiplies a general matrix by the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
void psormrq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );
```

```
void pdormrq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?ormrq` function overwrites the general real *m*-by-*n* distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
<i>trans</i> = 'T':	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where *Q* is a real orthogonal distributed matrix defined as the product of *k* elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by `p?gerqf`. *Q* is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

<i>side</i>	<p>(global)</p> <p>= 'L': Q or Q^T is applied from the left.</p> <p>= 'R': Q or Q^T is applied from the right.</p>
<i>trans</i>	<p>(global)</p> <p>= 'N', no transpose, Q is applied.</p> <p>= 'T', transpose, Q^T is applied.</p>
<i>m</i>	(global) The number of rows in the distributed matrix sub(C) ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix sub(C) ($n \geq 0$).
<i>k</i>	<p>(global) The number of elementary reflectors whose product defines the matrix Q. Constraints:</p> <p>If <i>side</i> = 'L', $m \geq k \geq 0$</p> <p>If <i>side</i> = 'R', $n \geq k \geq 0$.</p>
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a * LOCc(ja+m-1)$ if <i>side</i> = 'L', and $lld_a * LOCc(ja+n-1)$ if <i>side</i> = 'R'.</p> <p>The i-th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gerqf</code> in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.</p>
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>tau</i>	<p>(local)</p> <p>Array of size $LOCc(ja+k-1)$.</p> <p>Contains the scalar factor $\tau[i]$ of elementary reflectors $H(i+1)$ as returned by <code>p?gerqf</code> ($0 \leq i < LOCc(ja+k-1)$). <i>tau</i> is tied to the distributed matrix A.</p>
<i>c</i>	<p>(local)</p> <p>Pointer into the local memory to an array of local size $lld_c * LOCc(jc+n-1)$.</p> <p>Contains the local pieces of the distributed matrix sub(C) to be factored.</p>
<i>ic, jc</i>	(global) The row and column indices in the global matrix C indicating the first row and the first column of the matrix sub(C), respectively.
<i>descc</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix C .
<i>work</i>	<p>(local)</p> <p>Workspace array of size of <i>lwork</i>.</p>

lwork(local or global) size of *work*, must be at least:If *side* = 'L',

```
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + max(mqa0 +
numroc(numroc(n+iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0,
lcmp), nqc0))*mb_a) + mb_a*mb_a
```

else if *side* = 'R',

```
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a) + mb_a*mb_a
```

end if

where

```
lcmp = lcm/NPROW with lcm = ilcm (NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),
```

NOTEmod(*x*, *y*) is the integer remainder of *x*/*y*.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxxerbla.

Output Parameters*c*Overwritten by the product $Q * \text{sub}(C)$, or $Q' * \text{sub}(C)$, or $\text{sub}(C) * Q'$, or $\text{sub}(C) * Q$ *work*[0]On exit *work*[0] contains the minimum value of *lwork* required for optimum performance.*info*

(global)

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmrq

Multiplies a general matrix by the unitary matrix Q of the RQ factorization formed by p?gerqf.

Syntax

```
void pcunmrq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunmrq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

This function overwrites the general complex m -by- n distributed matrix sub (C) = $C(ic:ic+m-1,jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * sub(C)$	$sub(C) * Q$
$trans = 'C':$	$Q^H * sub(C)$	$sub(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(1)' H(2)' \dots H(k)'$$

as returned by p?gerqf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) = $'L'$: Q or Q^H is applied from the left. = $'R'$: Q or Q^H is applied from the right.
$trans$	(global) = $'N'$, no transpose, Q is applied. = $'C'$, conjugate transpose, Q^H is applied.
m	(global) The number of rows in the distributed matrix sub(C) , ($m \geq 0$) .
n	(global) The number of columns in the distributed matrix sub(C), ($n \geq 0$) .

<i>k</i>	<p>(global) The number of elementary reflectors whose product defines the matrix <i>Q</i>. Constraints:</p> <p>If <i>side</i> = 'L', $m \geq k \geq 0$</p> <p>If <i>side</i> = 'R', $n \geq k \geq 0$.</p>
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a * LOCC(ja+m-1)$ if <i>side</i> = 'L', and $lld_a * LOCC(ja+n-1)$ if <i>side</i> = 'R'. The <i>i</i>-th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gerqf</code> in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.</p>
<i>ia, ja</i>	<p>(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.</p>
<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p>
<i>tau</i>	<p>(local)</p> <p>Array of size $LOCC(ja+k-1)$.</p> <p>Contains the scalar factor $\tau[i]$ of elementary reflectors $H(i+1)$ as returned by <code>p?gerqf</code> ($0 \leq i < LOCC(ja+k-1)$). <i>tau</i> is tied to the distributed matrix <i>A</i>.</p>
<i>c</i>	<p>(local)</p> <p>Pointer into the local memory to an array of local size $lld_c * LOCC(jc+n-1)$.</p> <p>Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.</p>
<i>ic, jc</i>	<p>(global) The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the submatrix <i>C</i>, respectively.</p>
<i>descc</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>C</i>.</p>
<i>work</i>	<p>(local)</p> <p>Workspace array of size of <i>lwork</i>.</p>
<i>lwork</i>	<p>(local or global) size of <i>work</i>, must be at least:</p> <p>If <i>side</i> = 'L',</p> $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(n + iroffa, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0)) * mb_a) + mb_a * mb_a)$ <p>else if <i>side</i> = 'R',</p> $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a) + mb_a * mb_a$ <p>end if</p> <p>where</p> $lcmp = lcm / NPROW \text{ with } lcm = ilcm(NPROW, NPCOL),$ $iroffa = \text{mod}(ia - 1, mb_a),$

```

icoffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(m+icoffa, nb_a, MYCOL, iacol, NPCOL),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

`mod(x,y)` is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q^* \text{sub}(C)$ or $Q^* \text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?tzrzf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax

```

void pstzrzf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdtzrzf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

```

```
void pztzrzf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );
```

```
void pztzrzf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- mkl_scalapack.h

Description

The `pztzrzf` function reduces the m -by- n ($m \leq n$) real/complex upper trapezoidal matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix A is factored as

$$A = (R \ 0) * Z,$$

where Z is an n -by- n orthogonal/unitary matrix and R is an m -by- m upper triangular matrix.

Input Parameters

<i>m</i>	(global) The number of rows in the matrix $\text{sub}(A)$; ($m \geq 0$).
<i>n</i>	(global) The number of columns in the matrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$. Contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least $lwork \geq mb_a * (mp0 + nq0 + mb_a)$, where $iroff = \text{mod}(ia-1, mb_a),$ $icoff = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mp0 = \text{numroc}(m + iroff, mb_a, MYROW, iarow, NPROW),$ $nq0 = \text{numroc}(n + icoff, nb_a, MYCOL, iacol, NPCOL)$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	On exit, the leading m -by- m upper triangular part of <code>sub(A)</code> contains the upper triangular matrix R , and elements $m+1$ to n of the first m rows of <code>sub(A)</code> , with the array <code>tau</code> , represent the orthogonal/unitary matrix Z as a product of m elementary reflectors.
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>tau</code>	(local) Array of size <code>LOCr(ia+m-1)</code> . Contains the scalar factor of elementary reflectors. <code>tau</code> is tied to the distributed matrix A .
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

Application Notes

The factorization is obtained by the Householder's method. The k -th transformation matrix, $Z(k)$, which is or whose conjugate transpose is used to introduce zeros into the $(m - k + 1)$ -th row of `sub(A)`, is given in the form

$$Z(k) = \begin{bmatrix} i & 0 \\ 0 & T(k) \end{bmatrix}$$

where

$$T(k) = i - \tau u(k) u(k)',$$

$$u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

`tau` is a scalar and $Z(k)$ is an $(n - m)$ element vector. `tau` and $Z(k)$ are chosen to annihilate the elements of the k -th row of `sub(A)`. The scalar `tau` is returned in the k -th element of `tau`, indexed $k-1$, and the vector $u(k)$ in the k -th row of `sub(A)`, such that the elements of $Z(k)$ are in $a(k, m + 1), \dots, a(k, n)$. The elements of R are returned in the upper triangular part of `sub(A)`. Z is given by

$Z = Z(1) * Z(2) * \dots * Z(m)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormrz

Multiplies a general matrix by the orthogonal matrix from a reduction to upper triangular form formed by p?tzzrf.

Syntax

```
void psormrz (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_INT
 *l , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c ,
 MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , float *work , MKL_INT *lwork , MKL_INT
 *info );
```

```
void pdormrz (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_INT
 *l , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c ,
 MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , double *work , MKL_INT *lwork , MKL_INT
 *info );
```

Include Files

- mkl_scalapack.h

Description

This function overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$Q = H(1) H(2) \dots H(k)$

as returned by [p?tzzrf](#). Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) = $'L'$: Q or Q^T is applied from the left. = $'R'$: Q or Q^T is applied from the right.
$trans$	(global) = $'N'$, no transpose, Q is applied. = $'T'$, transpose, Q^T is applied.
m	(global) The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
n	(global) The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
k	(global) The number of elementary reflectors whose product defines the matrix Q . Constraints:

	<p>If $side = 'L', m \geq k \geq 0$</p> <p>If $side = 'R', n \geq k \geq 0$.</p>
<i>l</i>	<p>(global)</p> <p>The columns of the distributed matrix sub(<i>A</i>) containing the meaningful part of the Householder reflectors.</p> <p>If $side = 'L', m \geq l \geq 0$</p> <p>If $side = 'R', n \geq l \geq 0$.</p>
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a * LOCc(ja+m-1)$ if $side = 'L'$, and $lld_a * LOCc(ja+n-1)$ if $side = 'R'$, where $lld_a \geq \max(1, LOCr(ia+k-1))$.</p> <p>The <i>i</i>-th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?tzrzf</code> in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.</p>
<i>ia, ja</i>	<p>(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.</p>
<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p>
<i>tau</i>	<p>(local)</p> <p>Array of size $LOCc(ia+k-1)$.</p> <p>Contains the scalar factor $\tau[i]$ of elementary reflectors $H(i+1)$ as returned by <code>p?tzrzf</code> ($0 \leq i < LOCc(ia+k-1)$). <i>tau</i> is tied to the distributed matrix <i>A</i>.</p>
<i>c</i>	<p>(local)</p> <p>Pointer into the local memory to an array of local size $lld_c * LOCc(jc+n-1)$.</p> <p>Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.</p>
<i>ic, jc</i>	<p>(global) The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the submatrix <i>C</i>, respectively.</p>
<i>descc</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>C</i>.</p>
<i>work</i>	<p>(local)</p> <p>Workspace array of size of <i>lwork</i>.</p>
<i>lwork</i>	<p>(local or global) size of <i>work</i>, must be at least:</p> <p>If $side = 'L'$,</p> $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(n + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0)) * mb_a) + mb_a * mb_a)$ <p>else if $side = 'R'$,</p>

```
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a) + mb_a*mb_a
end if
```

where

```
lcmp = lcm/NPROW with lcm = ilcm (NPROW, NPCOL),
iroffa = mod(ia-1, mb_a), icoffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),
```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork` = -1, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q \cdot \text{sub}(C)$, or $Q' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q'$, or $\text{sub}(C) \cdot Q$
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i \cdot 100 + j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmrz

Multiplies a general matrix by the unitary transformation matrix from a reduction to upper triangular form determined by p?tzzrf.

Syntax

```
void pcunmrz (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_INT
 *l , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
 MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
 MKL_INT *lwork , MKL_INT *info );
```

```
void pzunmrz (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_INT
 *l , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16
 *tau , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16
 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

This function overwrites the general complex m -by- n distributed matrix sub (C) = $C(ic:ic+m-1,jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * sub(C)$	$sub(C) * Q$
$trans = 'C':$	$Q^H * sub(C)$	$sub(C) * Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(1)' H(2)' \dots H(k)'$$

as returned by pztzzrf/pztzzrf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) = $'L'$: Q or Q^H is applied from the left. = $'R'$: Q or Q^H is applied from the right.
$trans$	(global) = $'N'$, no transpose, Q is applied. = $'C'$, conjugate transpose, Q^H is applied.
m	(global) The number of rows in the distributed matrix sub(C), ($m \geq 0$).
n	(global) The number of columns in the distributed matrix sub(C), ($n \geq 0$).
k	(global) The number of elementary reflectors whose product defines the matrix Q . Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.

<i>l</i>	<p>(global) The columns of the distributed matrix sub(A) containing the meaningful part of the Householder reflectors.</p> <p>If <i>side</i> = 'L', $m \geq l \geq 0$</p> <p>If <i>side</i> = 'R', $n \geq l \geq 0$.</p>
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a * LOCC(ja+m-1)$ if <i>side</i> = 'L', and $lld_a * LOCC(ja+n-1)$ if <i>side</i> = 'R', where $lld_a \geq \max(1, LOCr(ja+k-1))$. The <i>i</i>-th row of the matrix stored in <i>amust</i> contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gerqf in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. $A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.</p>
<i>ia, ja</i>	<p>(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.</p>
<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p>
<i>tau</i>	<p>(local)</p> <p>Array of size $LOCC(ia+k-1)$.</p> <p>Contains the scalar factor $\tau[i]$ of elementary reflectors $H(i+1)$ as returned by p?gerqf ($0 \leq i < LOCC(ia+k-1)$). <i>tau</i> is tied to the distributed matrix <i>A</i>.</p>
<i>c</i>	<p>(local)</p> <p>Pointer into the local memory to an array of local size $lld_c * LOCC(jc+n-1)$.</p> <p>Contains the local pieces of the distributed matrix sub(C) to be factored.</p>
<i>ic, jc</i>	<p>(global) The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the submatrix <i>C</i>, respectively.</p>
<i>descc</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>C</i>.</p>
<i>work</i>	<p>(local)</p> <p>Workspace array of size <i>lwork</i>.</p>
<i>lwork</i>	<p>(local or global) size of <i>work</i>, must be at least:</p> <p>If <i>side</i> = 'L',</p> $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(n + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0)) * mb_a + mb_a * mb_a)$ <p>else if <i>side</i> = 'R',</p> $lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a + mb_a * mb_a)$ <p>end if</p> <p>where</p> $lcmp = lcm / NPROW \text{ with } lcm = ilcm(NPROW, NPCOL),$

```

iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(m+icoffa, nb_a, MYCOL, iacol, NPCOL),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

`mod(x,y)` is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q'^* \text{sub}(C)$, or $\text{sub}(C)Q'$, or $\text{sub}(C)Q$
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ggqrf

Computes the generalized QR factorization.

Syntax

```

void psggqrf (MKL_INT *n , MKL_INT *m , MKL_INT *p , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *taua , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , float *taub , float *work , MKL_INT *lwork , MKL_INT *info );

```

```

void pdggqrf (MKL_INT *n , MKL_INT *m , MKL_INT *p , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *taua , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , double *taub , double *work , MKL_INT *lwork , MKL_INT *info );

void pcggqrf (MKL_INT *n , MKL_INT *m , MKL_INT *p , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *taua , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *taub , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzggqrf (MKL_INT *n , MKL_INT *m , MKL_INT *p , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *taua , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *taub , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );

```

Include Files

- mkl_scalapack.h

Description

The `p?ggqrf` function forms the generalized *QR* factorization of an n -by- m matrix

$\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+m-1)$

and an n -by- p matrix

$\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+p-1):$

as

$\text{sub}(A) = Q \cdot R, \text{sub}(B) = Q \cdot T \cdot Z,$

where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

If $n \geq m$

$$R = \begin{pmatrix} R_{11} & \\ & 0 \end{pmatrix} \begin{matrix} m \\ n - m \\ m \end{matrix}$$

or if $n < m$

$$R = \begin{pmatrix} R_{11} & R_{12} \\ & \end{pmatrix} \begin{matrix} n \\ n \\ m - n \end{matrix}$$

where R_{11} is upper triangular, and

$$T = \begin{pmatrix} 0 & T_{12} \\ & \end{pmatrix} \begin{matrix} n, \text{ if } n \leq p, \\ p - n & n \end{matrix}$$

$$\text{or } T = \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \begin{pmatrix} n-p \\ p \end{pmatrix}, \text{ if } n > p,$$

p

where T_{12} or T_{21} is an upper triangular matrix.

In particular, if $\text{sub}(B)$ is square and nonsingular, the *GQR* factorization of $\text{sub}(A)$ and $\text{sub}(B)$ implicitly gives the *QR* factorization of $\text{inv}(\text{sub}(B)) * \text{sub}(A)$:

$$\text{inv}(\text{sub}(B)) * \text{sub}(A) = Z^H * (\text{inv}(T) * R)$$

Input Parameters

<i>n</i>	(global) The number of rows in the distributed matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
<i>m</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$ ($m \geq 0$).
<i>p</i>	The number of columns in the distributed matrix $\text{sub}(B)$ ($p \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size $\text{ld}_a * \text{LOCc}(ja+m-1)$. Contains the local pieces of the n -by- m matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>b</i>	(local) Pointer into the local memory to an array of size $\text{ld}_b * \text{LOCc}(jb+p-1)$. Contains the local pieces of the n -by- p matrix $\text{sub}(B)$ to be factored.
<i>ib, jb</i>	(global) The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B , respectively.
<i>descb</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix B .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) Size of <i>work</i> , must be at least $lwork \geq \max(nb_a * (npa0 + mqa0 + nb_a), \max((nb_a * (nb_a - 1)) / 2, (pqb0 + npb0) * nb_a) + nb_a * nb_a, mb_b * (npb0 + pqb0 + mb_b)),$ where $iroffa = \text{mod}(ia-1, mb_A),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}),$ $iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}),$


```

npa0 = numroc (n+iroffa, mb_a, MYROW, iarow, NPROW),
mqa0 = numroc (m+icoffa, nb_a, MYCOL, iacol, NPCOL)
iroffb = mod(ib-1, mb_b),
icoffb = mod(jb-1, nb_b),
ibrow = indxg2p(ib, mb_b, MYROW, rsrc_b, NPROW),
ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL),
npb0 = numroc (n+iroffa, mb_b, MYROW, Ibrow, NPROW),
pqb0 = numroc (m+icoffb, nb_b, MYCOL, ibcol, NPCOL)

```

NOTE

`mod(x, y)` is the integer remainder of x/y .

and `numroc`, `indxg2p` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

`a`

On exit, the elements on and above the diagonal of sub (A) contain the $\min(n, m)$ -by- m upper trapezoidal matrix R (R is upper triangular if $n \geq m$); the elements below the diagonal, with the array `taua`, represent the orthogonal/unitary matrix Q as a product of $\min(n, m)$ elementary reflectors. (See Application Notes below).

`taua, taub`

(local)

Arrays of size `LOCc(ja+min(n,m)-1)` for `taua` and `LOCr(ib+n-1)` for `taub`.

The array `taua` contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q . `taua` is tied to the distributed matrix A . (See Application Notes below).

The array `taub` contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z . `taub` is tied to the distributed matrix B . (See Application Notes below).

`work[0]`

On exit `work[0]` contains the minimum value of `lwork` required for optimum performance.

`info`

(global)

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then `info = -(i*100+j)`; if the i -th argument is a scalar and had an illegal value, then `info = -i`.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(j_a) * H(j_a+1) * \dots * H(j_a+k-1),$$

where $k = \min(n, m)$.

Each $H(i)$ has the form

$$H(i) = I - \tau_a u v^*$$

where τ_a is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i:ia+n-1, ja+i-1)$, and τ_a in $\tau_a[ja+i-2]$. To form Q explicitly, use ScaLAPACK function [p?orgqr/p?ungqr](#). To use Q to update another matrix, use ScaLAPACK function [p?ormqr/p?unmqr](#).

The matrix Z is represented as a product of elementary reflectors

$$Z = H(ib) * H(ib+1) * \dots * H(ib+k-1), \text{ where } k = \min(n, p).$$

Each $H(i)$ has the form

$$H(i) = I - \tau_b u v^*$$

where τ_b is a real/complex scalar, and v is a real/complex vector with $v(p-k+i+1:p) = 0$ and $v(p-k+i) = 1$; $v(1:p-k+i-1)$ is stored on exit in $B(ib+n-k+i-1, jb:jb+p-k+i-2)$, and τ_b in $\tau_b[ib+n-k+i-2]$. To form Z explicitly, use ScaLAPACK function [p?orgqr/p?ungrq](#). To use Z to update another matrix, use ScaLAPACK function [p?ormqr/p?unmrq](#).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ggrqf

Computes the generalized RQ factorization.

Syntax

```
void psggrqf (MKL_INT *m , MKL_INT *p , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *taua , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , float *taub , float *work , MKL_INT *lwork , MKL_INT *info );

void pdggrqf (MKL_INT *m , MKL_INT *p , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *taua , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , double *taub , double *work , MKL_INT *lwork , MKL_INT *info );

void pcggrqf (MKL_INT *m , MKL_INT *p , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *taua , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *taub , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzggrqf (MKL_INT *m , MKL_INT *p , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *taua , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *taub , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?ggrqf` function forms the generalized RQ factorization of an m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ and a p -by- n matrix $\text{sub}(B) = B(ib:ib+p-1, jb:jb+n-1)$:

$$\text{sub}(A) = R * Q, \text{ sub}(B) = Z * T * Q,$$

where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{pmatrix} 0 & R_{12} \\ R_{11} & 0 \end{pmatrix}, \text{ if } m \leq n,$$

or

$$R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix}, \text{ if } m > n$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix}, \text{ if } p \geq n$$

or

$$T = \begin{pmatrix} T_{11} & T_{12} \\ 0 & 0 \end{pmatrix}, \text{ if } p < n,$$

where T_{11} is upper triangular.

In particular, if $\text{sub}(B)$ is square and nonsingular, the GRQ factorization of $\text{sub}(A)$ and $\text{sub}(B)$ implicitly gives the RQ factorization of $\text{sub}(A) * \text{inv}(\text{sub}(B))$:

$$\text{sub}(A) * \text{inv}(\text{sub}(B)) = (R * \text{inv}(T)) * Z'$$

where $\text{inv}(\text{sub}(B))$ denotes the inverse of the matrix $\text{sub}(B)$, and Z' denotes the transpose (conjugate transpose) of matrix Z .

Input Parameters

m	(global) The number of rows in the distributed matrices $\text{sub}(A)$ ($m \geq 0$).
p	The number of rows in the distributed matrix $\text{sub}(B)$ ($p \geq 0$).
n	(global) The number of columns in the distributed matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
a	(local) Pointer into the local memory to an array of size $ld_a * LOCc(j_a + n - 1)$. Contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ to be factored.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) Pointer into the local memory to an array of size <i>lld_b*LOCc(jb+n-1)</i> . Contains the local pieces of the <i>p</i> -by- <i>n</i> matrix sub(<i>B</i>) to be factored.
<i>ib, jb</i>	(global) The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>work</i>	(local) Workspace array of size of <i>lwork</i> .
<i>lwork</i>	(local or global) Size of <i>work</i> , must be at least $lwork \geq \max(mb_a * (mpa0 + nqa0 + mb_a), \max((mb_a * (mb_a - 1)) / 2, (ppb0 + nqb0) * mb_a) + mb_a * mb_a, nb_b * (ppb0 + nqb0 + nb_b))$, where $iroffa = \text{mod}(ia - 1, mb_a),$ $icoffa = \text{mod}(ja - 1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$ $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$ $mpa0 = \text{numroc}(m + iroffa, mb_a, MYROW, iarow, NPROW),$ $nqa0 = \text{numroc}(m + icoffa, nb_a, MYCOL, iacol, NPCOL)$ $iroffb = \text{mod}(ib - 1, mb_b),$ $icoffb = \text{mod}(jb - 1, nb_b),$ $ibrow = \text{indxg2p}(ib, mb_b, MYROW, rsrc_b, NPROW),$ $ibcol = \text{indxg2p}(jb, nb_b, MYCOL, csrc_b, NPCOL),$ $ppb0 = \text{numroc}(p + iroffb, mb_b, MYROW, ibrow, NPROW),$ $nqb0 = \text{numroc}(n + icoffb, nb_b, MYCOL, ibcol, NPCOL)$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

and `numroc`, `indxg2p` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<code>a</code>	On exit, if $m \leq n$, the upper triangle of $A(ia:ia+m-1, ja+n-m:ja+n-1)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m-n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array <code>taua</code> , represent the orthogonal/unitary matrix Q as a product of $\min(n, m)$ elementary reflectors (see <i>Application Notes</i> below).
<code>taua, taub</code>	(local) Arrays of size $LOCr(ia+m-1)$ for <code>taua</code> and $LOCc(jb+\min(p, n)-1)$ for <code>taub</code> . The array <code>taua</code> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q . <code>taua</code> is tied to the distributed matrix A . (See <i>Application Notes</i> below). The array <code>taub</code> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z . <code>taub</code> is tied to the distributed matrix B . (See <i>Application Notes</i> below).
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ia) * H(ia+1) * \dots * H(ia+k-1),$$

where $k = \min(m, n)$.

Each $H(i)$ has the form

$$H(i) = I - \tau u v^* v'$$

where `taua` is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and `taua` in `taua[ia+m-k+i-2]`. To form Q explicitly, use ScaLAPACK function [p?orgqr/p?ungrq](#). To use Q to update another matrix, use ScaLAPACK function [p?ormqr/p?unmrq](#).

The matrix Z is represented as a product of elementary reflectors

$$Z = H(jb) * H(jb+1) * \dots * H(jb+k-1), \text{ where } k = \min(p, n).$$

Each $H(i)$ has the form

$$H(i) = I - \tau u v^* v'$$

where `taub` is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:p)$ is stored on exit in $B(ib+i:ib+p-1, jb+i-1)$, and `taub` in `taub[jb+i-2]`. To form Z explicitly, use ScaLAPACK function [p?orgqr/p?ungqr](#). To use Z to update another matrix, use ScaLAPACK function [p?ormqr/p?unmqr](#).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Symmetric Eigenvalue Problems: ScaLAPACK Computational Routines

To solve a symmetric eigenproblem with ScaLAPACK, you usually need to reduce the matrix to real tridiagonal form T and then find the eigenvalues and eigenvectors of the tridiagonal matrix T . ScaLAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in [Table "Computational Routines for Solving Symmetric Eigenproblems"](#).

There are different routines for symmetric eigenproblems, depending on whether you need eigenvalues only or eigenvectors as well, and on the algorithm used (either the QTQ algorithm, or bisection followed by inverse iteration).

Computational Routines for Solving Symmetric Eigenproblems

Operation	Dense symmetric/ Hermitian matrix	Orthogonal/unitary matrix	Symmetric tridiagonal matrix
Reduce to tridiagonal form $A = QTQ^H$	p?sytrd/p?hetrd		
Multiply matrix after reduction		p?ormtr/p?unmtr	
Find all eigenvalues and eigenvectors of a tridiagonal matrix T by a QTQ method			steqr2*
Find selected eigenvalues of a tridiagonal matrix T via bisection			p?stebz
Find selected eigenvectors of a tridiagonal matrix T by inverse iteration			p?stein

* This routine is described as part of auxiliary ScaLAPACK routines.

[p?syngst](#)

Reduces a complex Hermitian-definite generalized eigenproblem to standard form.

Syntax

```
void pssyngst (const MKL_INT* ibtype, const char* uplo, const MKL_INT* n, float* a,
const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const float* b, const
MKL_INT* ib, const MKL_INT* jb, const MKL_INT* descb, float* scale, float* work, const
MKL_INT* lwork, MKL_INT* info);
```

```
void pdsyngst (const MKL_INT* ibtype, const char* uplo, const MKL_INT* n, double* a,
const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const double* b, const
MKL_INT* ib, const MKL_INT* jb, const MKL_INT* descb, double* scale, double* work,
const MKL_INT* lwork, MKL_INT* info);
```

Include Files

- `mk1_scalapack.h`

Description

`p?syngst` reduces a complex Hermitian-definite generalized eigenproblem to standard form.

`p?syngst` performs the same function as `p?hegst`, but is based on rank 2K updates, which are faster and more scalable than triangular solves (the basis of `p?syngst`).

`p?syngst` calls `p?hegst` when `uplo='U'`, hence `p?hengst` provides improved performance only when `uplo='L'`, `ibtype=1`.

`p?syngst` also calls `p?hegst` when insufficient workspace is provided, hence `p?syngst` provides improved performance only when $lwork \geq 2 * NP0 * NB + NQ0 * NB + NB * NB$

In the following `sub(A)` denotes $A(ia:ia+n-1, ja:ja+n-1)$ and `sub(B)` denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If `ibtype = 1`, the problem is $sub(A) * x = \lambda * sub(B) * x$, and `sub(A)` is overwritten by $inv(U^H) * sub(A) * inv(U)$ or $inv(L) * sub(A) * inv(L^H)$

If `ibtype = 2` or `3`, the problem is $sub(A) * sub(B) * x = \lambda * x$ or $sub(B) * sub(A) * x = \lambda * x$, and `sub(A)` is overwritten by $U * sub(A) * U^H$ or $L^H * sub(A) * L$.

`sub(B)` must have been previously factorized as $U^H * U$ or $L * L^H$ by `p?potrf`.

Input Parameters

<code>ibtype</code>	(global) = 1: compute $inv(U^H) * sub(A) * inv(U)$ or $inv(L) * sub(A) * inv(L^H)$; = 2 or 3: compute $U * sub(A) * U^H$ or $L^H * sub(A) * L$.
<code>uplo</code>	(global) = 'U': Upper triangle of <code>sub(A)</code> is stored and <code>sub(B)</code> is factored as $U^H * U$; = 'L': Lower triangle of <code>sub(A)</code> is stored and <code>sub(B)</code> is factored as $L * L^H$.
<code>n</code>	(global) The order of the matrices <code>sub(A)</code> and <code>sub(B)</code> . $n \geq 0$.
<code>a</code>	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$. On entry, this array contains the local pieces of the n -by- n Hermitian distributed matrix <code>sub(A)</code> . If <code>uplo = 'U'</code> , the leading n -by- n upper triangular part of <code>sub(A)</code> contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <code>uplo = 'L'</code> , the leading n -by- n lower triangular part of <code>sub(A)</code> contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.
<code>ia</code>	(global) A's global row index, which points to the beginning of the submatrix which is to be operated on.
<code>ja</code>	(global) A's global column index, which points to the beginning of the submatrix which is to be operated on.
<code>desca</code>	(global and local) Array of size <code>dlen_</code> . The array descriptor for the distributed matrix <code>A</code> .
<code>b</code>	(local) Pointer into the local memory to an array of size $lld_b * LOCC(jb+n-1)$.

On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of $\text{sub}(B)$, as returned by `p?potrf`.

<i>ib</i>	(global) <i>B</i> 's global row index, which points to the beginning of the submatrix which is to be operated on.
<i>jb</i>	(global) <i>B</i> 's global column index, which points to the beginning of the submatrix which is to be operated on.
<i>descb</i>	(global and local) Array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>work</i>	(local) Array, size (<i>lwork</i>)
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq \text{MAX}(NB * (NP0 + 1), 3 * NB)$ When <i>ibtype</i> = 1 and <i>uplo</i> = 'L', <code>p?syngst</code> provides improved performance when $lwork \geq 2 * NP0 * NB + NQ0 * NB + NB * NB$, where $NB = mb_a = nb_a$, $NP0 = \text{numroc}(n, NB, 0, 0, NPROW)$, $NQ0 = \text{numroc}(n, NB, 0, 0, NPROW)$, <code>numroc</code> is a ScaLAPACK tool functions <i>MYROW</i> , <i>MYCOL</i> , <i>NPROW</i> and <i>NPCOL</i> can be determined by calling the subroutine <code>blacs_gridinfo</code> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>p?xerbla</code> .

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, the transformed matrix, stored in the same format as $\text{sub}(A)$.
<i>scale</i>	(global) Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, <i>scale</i> is always returned as 1.0, it is returned here to allow for future enhancement.
<i>work</i>	(local) Array, size (<i>lwork</i>)

On exit, `work[0]` returns the minimal and optimal `lwork`.

`info`

(global)

= 0: successful exit

< 0: If the i -th argument is an array and the j -th entry had an illegal value, then `info` = $-(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then `info` = $-i$.

p?syntdr

Reduces a real symmetric matrix to symmetric tridiagonal form.

Syntax

```
void pssyntdr (const char* uplo, const MKL_INT* n, float* a, const MKL_INT* ia, const
MKL_INT* ja, const MKL_INT* desca, float* d, float* e, float* tau, float* work, const
MKL_INT* lwork, MKL_INT* info);
```

```
void pdsyntdr (const char* uplo, const MKL_INT* n, double* a, const MKL_INT* ia, const
MKL_INT* ja, const MKL_INT* desca, double* d, double* e, double* tau, double* work,
const MKL_INT* lwork, MKL_INT* info);
```

Include Files

- `mkl_scalapack.h`

Description

`p?syntdr` is a prototype version of `p?sytrd` which uses tailored codes (either the serial, `?sytrd`, or the parallel code, `p?sytrdr`) when the workspace provided by the user is adequate.

`p?syntdr` reduces a real symmetric matrix `sub(A)` to symmetric tridiagonal form `T` by an orthogonal similarity transformation:

$Q' * \text{sub}(A) * Q = T$, where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Features

`p?syntdr` is faster than `p?sytrd` on almost all matrices, particularly small ones (i.e. $n < 500 * \text{sqrt}(P)$), provided that enough workspace is available to use the tailored codes.

The tailored codes provide performance that is essentially independent of the input data layout.

The tailored codes place no restrictions on `ia`, `ja`, `MB` or `NB`. At present, `ia`, `ja`, `MB` and `NB` are restricted to those values allowed by `p?hetdr` to keep the interface simple (see the Application Notes section for more information about the restrictions).

Input Parameters

`uplo`

(global)

Specifies whether the upper or lower triangular part of the symmetric matrix `sub(A)` is stored:

= 'U': Upper triangular

= 'L': Lower triangular

`n`

(global)

The number of rows and columns to be operated on, i.e. the order of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.

a

(local)

Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$.

On entry, this array contains the local pieces of the symmetric distributed matrix $\text{sub}(A)$. If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia

(global)

The row index in the global array *a* indicating the first row of $\text{sub}(A)$.

ja

(global)

The column index in the global array *a* indicating the first column of $\text{sub}(A)$.

desca

(global and local)

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

work

(local)

Array, size (*lwork*)

lwork

(local or global)

The size of the array *work*.

lwork is local input and must be at least $lwork \geq \text{MAX}(NB * (NP + 1), 3 * NB)$

For optimal performance, greater workspace is needed, i.e.

$lwork \geq 2 * (ANB + 1) * (4 * NPS + 2) + (NPS + 4) * NPS$

$ANB = \text{pjlaenv}(ICTXT, 3, 'p?sytttrd', 'L', 0, 0, 0, 0)$

$ICTXT = \text{desca}(ctxt_)$

$SQNPC = \text{INT}(\text{sqrt}(\text{REAL}(NPROW * NPCOL)))$

numroc is a ScaLAPACK tool function.

pjlaenv is a ScaLAPACK environmental inquiry function.

NPROW and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

Output Parameters

a

On exit, if $uplo = 'U'$, the diagonal and first superdiagonal of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors; if $uplo = 'L'$, the diagonal and first subdiagonal

of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements below the first subdiagonal, with the array τ , represent the orthogonal matrix Q as a product of elementary reflectors. See **Further Details**.

d	<p>(local) Array, size $\text{LOCc}(ja+n-1)$</p> <p>The diagonal elements of the tridiagonal matrix T: $d(i) = A(i,i)$. d is tied to the distributed matrix A.</p>
e	<p>(local) Array, size $\text{LOCc}(ja+n-1)$ if $uplo = 'U'$, $\text{LOCc}(ja+n-2)$ otherwise.</p> <p>The off-diagonal elements of the tridiagonal matrix T: $e(i) = A(i,i+1)$ if $uplo = 'U'$, $e(i) = A(i+1,i)$ if $uplo = 'L'$. e is tied to the distributed matrix A.</p>
τ	<p>(local) Array, size $\text{LOCc}(ja+n-1)$.</p> <p>This array contains the scalar factors τ of the elementary reflectors. τ is tied to the distributed matrix A.</p>
$work$	<p>(local) Array, size $(lwork)$</p> <p>On exit, $work[0]$ returns the optimal $lwork$.</p>
$info$	<p>(global)</p> <p>= 0: successful exit</p> <p>< 0: If the i-th argument is an array and the j-th entry had an illegal value, then $info = -(i*100+j)$, if the i-th argument is a scalar and had an illegal value, then $info = -i$.</p>

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) \dots H(2) H(1).$$

Each $H(i)$ has the form

$H(i) = I - \tau * v * v'$, where τ is a complex scalar, and v is a complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τ in $\tau(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) H(2) \dots H(n-1).$$

Each $H(i)$ has the form

$H(i) = I - \tau * v * v'$, where τ is a complex scalar, and v is a complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

The contents of $\text{sub}(A)$ on exit are illustrated by the following examples with $n = 5$:

if $uplo = 'U'$:

$$\begin{pmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v3 \\ & & & d & e \\ & & & & d \end{pmatrix}$$

if *uplo* = 'L':

$$\begin{pmatrix} d \\ e & d \\ v1 & e & d \\ v1 & v2 & e & d \\ v1 & v2 & v3 & e & d \end{pmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *T*, and *vi* denotes an element of the vector defining *H(i)*.

Alignment requirements

The distributed submatrix sub(*A*) must verify some alignment properties, namely the following expression should be true:

(*mb_a* = *nb_a* and *IROFFA* = *ICOFFA* and *IROFFA* = 0) with *IROFFA* = mod(*ia*-1, *mb_a*), and *ICOFFA* = mod(*ja*-1, *nb_a*).

p?sytrd

Reduces a symmetric matrix to real symmetric tridiagonal form by an orthogonal similarity transformation.

Syntax

```
void pssytrd (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *d , float *e , float *tau , float *work , MKL_INT *lwork , MKL_INT
*info );
```

```
void pdsytrd (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *d , double *e , double *tau , double *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- mkl_scalapack.h

Description

The p?sytrd function reduces a real symmetric matrix sub(*A*) to symmetric tridiagonal form *T* by an orthogonal similarity transformation:

$$Q^* \text{sub}(A) Q = T,$$

where sub(*A*) = *A*(*ia:ia*+*n*-1,*ja:ja*+*n*-1).

Input Parameters

uplo (global)
Specifies whether the upper or lower triangular part of the symmetric matrix sub(*A*) is stored:
If *uplo* = 'U', upper triangular

	If <code>uplo = 'L'</code> , lower triangular
<code>n</code>	(global) The order of the distributed matrix <code>sub(A)</code> ($n \geq 0$).
<code>a</code>	(local) Pointer into the local memory to an array of size <code>ld_a*LOCc(ja+n-1)</code> . On entry, this array contains the local pieces of the symmetric distributed matrix <code>sub(A)</code> . If <code>uplo = 'U'</code> , the leading n -by- n upper triangular part of <code>sub(A)</code> contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <code>uplo = 'L'</code> , the leading n -by- n lower triangular part of <code>sub(A)</code> contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. See <i>Application Notes</i> below.
<code>ia, ja</code>	(global) The row and column indices in the global matrix <code>A</code> indicating the first row and the first column of the submatrix <code>A</code> , respectively.
<code>desca</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix <code>A</code> .
<code>work</code>	(local) Workspace array of size <code>lwork</code> .
<code>lwork</code>	(local or global) size of <code>work</code> , must be at least: $lwork \geq \max(NB * (np + 1), 3 * NB),$ where $NB = mb_a = nb_a$, $np = \text{numroc}(n, NB, MYROW, iarow, NPROW),$ $iarow = \text{indxg2p}(ia, NB, MYROW, rsrc_a, NPROW).$ <code>indxg2p</code> and <code>numroc</code> are ScaLAPACK tool functions; <code>MYROW</code> , <code>MYCOL</code> , <code>NPROW</code> and <code>NPCOL</code> can be determined by calling the function <code>blacs_gridinfo</code> . If <code>lwork = -1</code> , then <code>lwork</code> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pxerbla</code> .

Output Parameters

<code>a</code>	On exit, if <code>uplo = 'U'</code> , the diagonal and first superdiagonal of <code>sub(A)</code> are overwritten by the corresponding elements of the tridiagonal matrix <code>T</code> , and the elements above the first superdiagonal, with the array <code>tau</code> , represent the orthogonal matrix <code>Q</code> as a product of elementary reflectors; if <code>uplo = 'L'</code> , the diagonal and first subdiagonal of <code>sub(A)</code> are overwritten by the corresponding elements of the tridiagonal matrix <code>T</code> , and the elements below the first subdiagonal, with the array <code>tau</code> , represent the orthogonal matrix <code>Q</code> as a product of elementary reflectors. See <i>Application Notes</i> below.
<code>d</code>	(local) Arrays of size <code>LOCc(ja+n-1)</code> . The diagonal elements of the tridiagonal matrix <code>T</code> :

$d[i] = A(i+1, i+1), 0 \leq i < LOCC(ja+n-1).$

d is tied to the distributed matrix A .

e

(local)

Arrays of size $LOCC(ja+n-1)$ if $uplo = 'U'$, $LOCC(ja+n-2)$ otherwise.

The off-diagonal elements of the tridiagonal matrix T :

$e[i] = A(i+1, i+2), 0 \leq i < LOCC(ja+n-1)$ if $uplo = 'U'$,

$e[i] = A(i+2, i+1)$ if $uplo = 'L'$.

e is tied to the distributed matrix A .

tau

(local)

Arrays of size $LOCC(ja+n-1)$. This array contains the scalar factors of the elementary reflectors. tau is tied to the distributed matrix A .

$work[0]$

On exit $work[0]$ contains the minimum value of $lwork$ required for optimum performance.

$info$

(global)

$= 0$: the execution is successful.

< 0 : if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) \dots H(2) H(1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau u * v * v',$$

where τu is a real scalar, and v is a real vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τu in $\tau u[ja+i-2]$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) H(2) \dots H(n-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau u * v * v',$$

where τu is a real scalar, and v is a real vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τu in $\tau u[ja+i-2]$.

The contents of $sub(A)$ on exit are illustrated by the following examples with $n = 5$:

If $uplo = 'U'$:

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

If *uplo* = 'L':

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where *d* and *e* denote diagonal and off-diagonal elements of *T*, and *vi* denotes an element of the vector defining *H(i)*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormtr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to tridiagonal form determined by p?sytrd.

Syntax

```
void psormtr (char *side , char *uplo , char *trans , MKL_INT *m , MKL_INT *n , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );
```

```
void pdormtr (char *side , char *uplo , char *trans , MKL_INT *m , MKL_INT *n , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

This function overwrites the general real distributed *m*-by-*n* matrix sub(*C*) = *C*(*ic:ic+m-1,jc:jc+n-1*) with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	<i>Q</i> *sub(<i>C</i>)	sub(<i>C</i>)* <i>Q</i>
<i>trans</i> = 'T':	<i>Q</i> ^T *sub(<i>C</i>)	sub(<i>C</i>)* <i>Q</i> ^T

where *Q* is a real orthogonal distributed matrix of order *nq*, with *nq* = *m* if *side* = 'L' and *nq* = *n* if *side* = 'R'.

Q is defined as the product of nq elementary reflectors, as returned by `p?sytrd`.

If `uplo = 'U'`, $Q = H(nq-1) \dots H(2) H(1)$;

If `uplo = 'L'`, $Q = H(1) H(2) \dots H(nq-1)$.

Input Parameters

<code>side</code>	(global) = <code>'L'</code> : Q or Q^T is applied from the left. = <code>'R'</code> : Q or Q^T is applied from the right.
<code>trans</code>	(global) = <code>'N'</code> , no transpose, Q is applied. = <code>'T'</code> , transpose, Q^T is applied.
<code>uplo</code>	(global) = <code>'U'</code> : Upper triangle of $A(ia:*, ja:*)$ contains elementary reflectors from <code>p?sytrd</code> ; = <code>'L'</code> : Lower triangle of $A(ia:*, ja:*)$ contains elementary reflectors from <code>p?sytrd</code>
<code>m</code>	(global) The number of rows in the distributed matrix sub(C) ($m \geq 0$).
<code>n</code>	(global) The number of columns in the distributed matrix sub(C) ($n \geq 0$).
<code>a</code>	(local) Pointer into the local memory to an array of size $lld_a * LOCc(ja+m-1)$ if <code>side = 'L'</code> , and $lld_a * LOCc(ja+n-1)$ if <code>side = 'R'</code> . Contains the vectors that define the elementary reflectors, as returned by <code>p?sytrd</code> . If <code>side='L'</code> , $lld_a \geq \max(1, LOCr(ia+m-1))$; If <code>side = 'R'</code> , $lld_a \geq \max(1, LOCr(ia+n-1))$.
<code>ia, ja</code>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<code>desca</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .
<code>tau</code>	(local) Array of size of <code>ltau</code> where if <code>side = 'L'</code> and <code>uplo = 'U'</code> , $ltau = LOCc(m_a)$, if <code>side = 'L'</code> and <code>uplo = 'L'</code> , $ltau = LOCc(ja+m-2)$, if <code>side = 'R'</code> and <code>uplo = 'U'</code> , $ltau = LOCc(n_a)$, if <code>side = 'R'</code> and <code>uplo = 'L'</code> , $ltau = LOCc(ja+n-2)$. $tau[i]$ must contain the scalar factor of the elementary reflector $H(i+1)$, as returned by <code>p?sytrd</code> ($0 \leq i < ltau$). tau is tied to the distributed matrix A .
<code>c</code>	(local)

Pointer into the local memory to an array of size $lld_c * LOCc(jc+n-1)$.
Contains the local pieces of the distributed matrix sub (C).

ic, jc (global) The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C, respectively.

desc (global and local) array of size *dlen_*. The array descriptor for the distributed matrix C.

work (local)
Workspace array of size *lwork*.

lwork (local or global) size of *work*, must be at least:

```

if uplo = 'U',
  iaa= ia; jaa= ja+1, icc= ic; jcc= jc;
else uplo = 'L',
  iaa= ia+1, jaa= ja;
If side = 'L',
  icc= ic+1; jcc= jc;
else icc= ic; jcc= jc+1;
end if
end if
If side = 'L',
mi= m-1; ni= n
lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) +
nb_a*nb_a
else
If side = 'R',
mi= m; ni = n-1;
lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 +
max(npa0+numroc(numroc(ni+icoffc, nb_a, 0, 0, NPCOL), nb_a,
0, 0, lcmq), mpc0))*nb_a)+ nb_a*nb_a
end if
where lcmq = lcm/NPCOL with lcm = ilcm(NPROW, NPCOL),
iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),
icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),

```

```
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),
```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`. If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pzhengst`.

Output Parameters

<code>c</code>	Overwritten by the product $Q \cdot \text{sub}(C)$, or $Q' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q'$, or $\text{sub}(C) \cdot Q$.
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i \cdot 100 + j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

pzhengst

Reduces a complex Hermitian-definite generalized eigenproblem to standard form.

Syntax

```
void pzhengst (const MKL_INT* ibtype, const char* uplo, const MKL_INT* n, MKL_Complex8*
a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const MKL_Complex8* b,
const MKL_INT* ib, const MKL_INT* jb, const MKL_INT* descb, float* scale, MKL_Complex8*
work, const MKL_INT* lwork, MKL_INT* info);
```

```
void pzhengst (const MKL_INT* ibtype, const char* uplo, const MKL_INT* n,
MKL_Complex16* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const
MKL_Complex16* b, const MKL_INT* ib, const MKL_INT* jb, const MKL_INT* descb, double*
scale, MKL_Complex16* work, const MKL_INT* lwork, MKL_INT* info);
```

Include Files

- `mkc_scaLapack.h`

Description

`p?hengst` reduces a complex Hermitian-definite generalized eigenproblem to standard form.

`p?hengst` performs the same function as `p?hegst`, but is based on rank 2K updates, which are faster and more scalable than triangular solves (the basis of `p?hengst`).

`p?hengst` calls `p?hegst` when `uplo='U'`, hence `p?hengst` provides improved performance only when `uplo='L'` and `ibtype=1`.

`p?hengst` also calls `p?hegst` when insufficient workspace is provided, hence `p?hengst` provides improved performance only when `lwork` is sufficient (as described in the parameter descriptions).

In the following `sub(A)` denotes the submatrix $A(ia:ia+n-1, ja:ja+n-1)$ and `sub(B)` denotes the submatrix $B(ib:ib+n-1, jb:jb+n-1)$.

If `ibtype = 1`, the problem is $\text{sub}(A) * x = \text{lambda} * \text{sub}(B) * x$, and `sub(A)` is overwritten by $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$ or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$

If `ibtype = 2` or `3`, the problem is $\text{sub}(A) * \text{sub}(B) * x = \text{lambda} * x$ or $\text{sub}(B) * \text{sub}(A) * x = \text{lambda} * x$, and `sub(A)` is overwritten by $U * \text{sub}(A) * U^H$ or $L^H * \text{sub}(A) * L$.

`sub(B)` must have been previously factorized as $U^H * U$ or $L * L^H$ by `p?potrf`.

Input Parameters

<code>ibtype</code>	(global) = 1: compute $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$ or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$; = 2 or 3: compute $U * \text{sub}(A) * U^H$ or $L^H * \text{sub}(A) * L$.
<code>uplo</code>	(global) = 'U': Upper triangle of <code>sub(A)</code> is stored and <code>sub(B)</code> is factored as $U^H * U$; = 'L': Lower triangle of <code>sub(A)</code> is stored and <code>sub(B)</code> is factored as $L * L^H$.
<code>n</code>	(global) The order of the matrices <code>sub(A)</code> and <code>sub(B)</code> . $n \geq 0$.
<code>a</code>	(local) Pointer into the local memory to an array of size <code>lld_a * LOCc(ja+n-1)</code> . On entry, this array contains the local pieces of the n -by- n Hermitian distributed matrix <code>sub(A)</code> . If <code>uplo = 'U'</code> , the leading n -by- n upper triangular part of <code>sub(A)</code> contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <code>uplo = 'L'</code> , the leading n -by- n lower triangular part of <code>sub(A)</code> contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.
<code>ia</code>	(global) Global row index of matrix A , which points to the beginning of the submatrix on which to operate.
<code>ja</code>	(global) Global column index of matrix A , which points to the beginning of the submatrix on which to operate.

<i>desca</i>	(global and local) Array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) Pointer into the local memory to an array of size <i>lld_b*LOCc(jb+n-1)</i> .
<i>ib</i>	(global) Global row index of matrix <i>B</i> , which points to the beginning of the submatrix on which to operate.
<i>jb</i>	(global) Global column index of matrix <i>B</i> , which points to the beginning of the submatrix on which to operate.
<i>descb</i>	(global and local) Array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>work</i>	(local) Array, size (<i>lwork</i>) On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>lwork</i>	(local) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq \text{MAX}(NB * (NPO + 1), 3 * NB)$. When <i>ibtype</i> = 1 and <i>uplo</i> = 'L', p?hengst provides improved performance when $lwork \geq 2 * NPO * NB + NQ0 * NB + NB * NB$, where $NB = mb_a = nb_a$, $NPO = \text{numroc}(n, NB, 0, 0, \text{NPROW})$, $NQ0 = \text{numroc}(n, NB, 0, 0, \text{NPROW})$, and <i>numroc</i> is a ScaLAPACK tool function. <i>MYROW</i> , <i>MYCOL</i> , <i>NPROW</i> and <i>NPCOL</i> can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>p?erbla</i> .

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, the transformed matrix, stored in the same format as sub(<i>A</i>).
<i>scale</i>	(global) Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. <i>scale</i> is always returned as 1.0.

work On exit, *work*[0] returns the minimal and optimal *lwork*.

info (global)

= 0: successful exit

< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i*100+j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

p?hentrdd

Reduces a complex Hermitian matrix to Hermitian tridiagonal form.

Syntax

```
void pchentrdd (const char* uplo, const MKL_INT* n, MKL_Complex8* a, const MKL_INT* ia,
const MKL_INT* ja, const MKL_INT* desca, float* d, float* e, MKL_Complex8* tau,
MKL_Complex8* work, const MKL_INT* lwork, float* rwork, const MKL_INT* lrwork, MKL_INT*
info);
```

```
void pzentrdd (const char* uplo, const MKL_INT* n, MKL_Complex16* a, const MKL_INT* ia,
const MKL_INT* ja, const MKL_INT* desca, double* d, double* e, MKL_Complex16* tau,
MKL_Complex16* work, const MKL_INT* lwork, double* rwork, const MKL_INT* lrwork,
MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

p?hentrdd is a prototype version of p?hetrdd which uses tailored codes (either the serial, ?hetrdd, or the parallel code, p?hettrdd) when adequate workspace is provided.

p?hentrdd reduces a complex Hermitian matrix sub(*A*) to Hermitian tridiagonal form *T* by an unitary similarity transformation:

$Q' * \text{sub}(A) * Q = T$, where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

p?hentrdd is faster than p?hetrdd on almost all matrices, particularly small ones (i.e. $n < 500 * \sqrt{P}$), provided that enough workspace is available to use the tailored codes.

The tailored codes provide performance that is essentially independent of the input data layout.

The tailored codes place no restrictions on *ia*, *ja*, MB or NB. At present, *ia*, *ja*, MB and NB are restricted to those values allowed by p?hetrdd to keep the interface simple (see the Application Notes section for more information about the restrictions).

Input Parameters

uplo (global)

Specifies whether the upper or lower triangular part of the Hermitian matrix sub(*A*) is stored:

= 'U': Upper triangular

= 'L': Lower triangular

n (global)

The number of rows and columns to be operated on, i.e. the order of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.

a

(local)

Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$.

On entry, this array contains the local pieces of the Hermitian distributed matrix $\text{sub}(A)$. If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia

(global)

The row index in the global array *a* indicating the first row of $\text{sub}(A)$.

ja

(global)

The column index in the global array *a* indicating the first column of $\text{sub}(A)$.

desca

(global and local)

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

work

(local)

Array, size (*lwork*)

lwork

(local or global)

The size of the array *work*.

lwork is local input and must be at least $lwork \geq \text{MAX}(NB * (NP + 1), 3 * NB)$.

For optimal performance, greater workspace is needed:

$lwork \geq 2 * (ANB + 1) * (4 * NPS + 2) + (NPS + 4) * NPS$

$ANB = \text{pjlaenv}(ICTXT, 3, 'p?hettrd', 'L', 0, 0, 0, 0)$

$ICTXT = \text{desca}(ctxt_)$

$SQNPC = \text{INT}(\text{sqrt}(\text{REAL}(NPROW * NPCOL)))$

$NPS = \text{MAX}(\text{numroc}(n, 1, 0, 0, SQNPC), 2 * ANB)$

numroc is a ScaLAPACK tool function.

pjlaenv is a ScaLAPACK environmental inquiry function.

NPROW and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

rwork

(local)

Array, size (*lrwork*)

lrwork

(local or global)

The size of the array *rwork*.

lwork is local input and must be at least $lwork \geq 1$.

For optimal performance, greater workspace is needed, i.e. $lwork \geq \text{MAX}(2 * n)$

Output Parameters

<i>a</i>	On exit, if <i>uplo</i> = 'U', the diagonal and first superdiagonal of sub(<i>A</i>) are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i> , and the elements above the first superdiagonal, with the array <i>tau</i> , represent the unitary matrix <i>Q</i> as a product of elementary reflectors; if <i>uplo</i> = 'L', the diagonal and first subdiagonal of sub(<i>A</i>) are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i> , and the elements below the first subdiagonal, with the array <i>tau</i> , represent the unitary matrix <i>Q</i> as a product of elementary reflectors. See Application Notes.
<i>d</i>	(local) Array, size $\text{LOCc}(ja+n-1)$ The diagonal elements of the tridiagonal matrix <i>T</i> : $d[i - 1] = A(i,i)$. <i>d</i> is tied to the distributed matrix <i>A</i> .
<i>e</i>	(local) Array, size $\text{LOCc}(ja+n-1)$ if <i>uplo</i> = 'U', $\text{LOCc}(ja+n-2)$ otherwise. The off-diagonal elements of the tridiagonal matrix <i>T</i> : $e[i - 1] = A(i,i+1)$ if <i>uplo</i> = 'U', $e[i - 1] = A(i+1,i)$ if <i>uplo</i> = 'L'. <i>e</i> is tied to the distributed matrix <i>A</i> .
<i>tau</i>	(local) Array, size $\text{LOCc}(ja+n-1)$. This array contains the scalar factors <i>tau</i> of the elementary reflectors. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	On exit, <i>work</i> [0] returns the optimal <i>lwork</i> .
<i>rwork</i>	On exit, <i>rwork</i> [0] returns the optimal <i>lwork</i> .
<i>info</i>	(global) = 0: successful exit < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = $-(i*100+j)$, if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

Application Notes

If *uplo* = 'U', the matrix *Q* is represented as a product of elementary reflectors

$$Q = H(n-1) \dots H(2) H(1).$$

Each *H*(*i*) has the form

$H(i) = I - \tau u * v * v'$, where *tau* is a complex scalar, and *v* is a complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2,ja+i)$, and *tau* in $\tau(ja+i-1)$.

If `uplo = 'L'`, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) H(2) \dots H(n-1).$$

Each $H(i)$ has the form

$H(i) = I - \tau v v'$, where τ is a complex scalar, and v is a complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τ in $\tau(ja+i-1)$.

The contents of `sub(A)` on exit are illustrated by the following examples with $n = 5$:

if `uplo = 'U'`:

$$\begin{pmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v3 \\ & & & d & e \\ & & & & d \end{pmatrix}$$

if `uplo = 'L'`:

$$\begin{pmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{pmatrix}$$

where d and e denote diagonal and off-diagonal elements of T , and v_i denotes an element of the vector defining $H(i)$.

Alignment requirements

The distributed submatrix `sub(A)` must verify some alignment properties, namely the following expression should be true:

$(mb_a = nb_a \text{ and } IROFFA = ICOFFA \text{ and } IROFFA = 0)$ with $IROFFA = \text{mod}(ia-1, mb_a)$, and $ICOFFA = \text{mod}(ja-1, nb_a)$.

p?hetrd

Reduces a Hermitian matrix to Hermitian tridiagonal form by a unitary similarity transformation.

Syntax

```
void pchetrdr (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *d , float *e , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
```

```
void pzhetrd (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *d , double *e , MKL_Complex16 *tau , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?hetrd` function reduces a complex Hermitian matrix `sub(A)` to Hermitian tridiagonal form T by a unitary similarity transformation:

$$Q^* \text{sub}(A) Q = T$$

where $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$.

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the Hermitian matrix $\text{sub}(A)$ is stored: If <i>uplo</i> = 'U', upper triangular If <i>uplo</i> = 'L', lower triangular
<i>n</i>	(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size $\text{lld_a} * \text{LOCc}(\text{ja}+n-1)$. On entry, this array contains the local pieces of the Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. (see <i>Application Notes</i> below).
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least: $\text{lwork} \geq \max(\text{NB} * (\text{np} + 1), 3 * \text{NB})$ where $\text{NB} = \text{mb_a} = \text{nb_a}$, $\text{np} = \text{numroc}(n, \text{NB}, \text{MYROW}, \text{iarow}, \text{NPROW})$, $\text{iarow} = \text{indxg2p}(\text{ia}, \text{NB}, \text{MYROW}, \text{rsrc_a}, \text{NPROW})$. <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>pxerbla</i> .

Output Parameters

<i>a</i>	On exit,
----------	----------

If `uplo = 'U'`, the diagonal and first superdiagonal of `sub(A)` are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements above the first superdiagonal, with the array `tau`, represent the unitary matrix Q as a product of elementary reflectors; if `uplo = 'L'`, the diagonal and first subdiagonal of `sub(A)` are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements below the first subdiagonal, with the array `tau`, represent the unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

`d`

(local)

Arrays of size `LOCc(ja+n-1)`. The diagonal elements of the tridiagonal matrix T :

$$d[i] = A(i+1, i+1), 0 \leq i < LOCc(ja+n-1).$$

`d` is tied to the distributed matrix A .

`e`

(local)

Arrays of size `LOCc(ja+n-1)` if `uplo = 'U'`; `LOCc(ja+n-2)` - otherwise.

The off-diagonal elements of the tridiagonal matrix T :

$$e[i] = A(i+1, i+2), 0 \leq i < LOCc(ja+n-1) \text{ if } uplo = 'U',$$

$$e[i] = A(i+2, i+1) \text{ if } uplo = 'L'.$$

`e` is tied to the distributed matrix A .

`tau`

(local)

Array of size `LOCc(ja+n-1)`. This array contains the scalar factors of the elementary reflectors. `tau` is tied to the distributed matrix A .

`work[0]`

On exit `work[0]` contains the minimum value of `lwork` required for optimum performance.

`info`

(global)

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then `info` = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then `info` = $-i$.

Application Notes

If `uplo = 'U'`, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) * \dots * H(2) * H(1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau u v^*,$$

where τ is a complex scalar, and v is a complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τ in $\tau[ja+i-2]$.

If `uplo = 'L'`, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(n-1).$$

Each $H(i)$ has the form

$$H(i) = i - \tau v v',$$

where τ is a complex scalar, and v is a complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τ in $\tau[ja+i-2]$.

The contents of $\text{sub}(A)$ on exit are illustrated by the following examples with $n = 5$:

If $\text{uplo} = 'U'$:

$$\begin{bmatrix} d & e & v2 & v3 & v4 \\ & d & e & v3 & v4 \\ & & d & e & v4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

If $\text{uplo} = 'L'$:

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v1 & e & d & & \\ v1 & v2 & e & d & \\ v1 & v2 & v3 & e & d \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of T , and v_i denotes an element of the vector defining $H(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmtr

Multiplies a general matrix by the unitary transformation matrix from a reduction to tridiagonal form determined by p?hetrd.

Syntax

```
void pcunmtr (char *side , char *uplo , char *trans , MKL_INT *m , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunmtr (char *side , char *uplo , char *trans , MKL_INT *m , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

This function overwrites the general complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'C':$	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix of order nq , with $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$.

Q is defined as the product of $nq-1$ elementary reflectors, as returned by `p?hetrd`.

If $uplo = 'U'$, $Q = H(nq-1) \dots H(2) H(1)$;

If $uplo = 'L'$, $Q = H(1) H(2) \dots H(nq-1)$.

Input Parameters

$side$	(global) $= 'L'$: Q or Q^H is applied from the left. $= 'R'$: Q or Q^H is applied from the right.
$trans$	(global) $= 'N'$, no transpose, Q is applied. $= 'C'$, conjugate transpose, Q^H is applied.
$uplo$	(global) $= 'U'$: Upper triangle of $A(ia:*, ja:*)$ contains elementary reflectors from <code>p?hetrd</code> ; $= 'L'$: Lower triangle of $A(ia:*, ja:*)$ contains elementary reflectors from <code>p?hetrd</code>
m	(global) The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
n	(global) The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
a	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja+m-1)$ if $side = 'L'$, and $lld_a * LOCC(ja+n-1)$ if $side = 'R'$. Contains the vectors which define the elementary reflectors, as returned by <code>p?hetrd</code> . If $side = 'L'$, $lld_a \geq \max(1, LOCr(ia+m-1))$; If $side = 'R'$, $lld_a \geq \max(1, LOCr(ia+n-1))$.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
tau	(local)

Array of size of *ltau* where

```
If side = 'L' and uplo = 'U', ltau = LOCc(m_a),
if side = 'L' and uplo = 'L', ltau = LOCc(ja+m-2),
if side = 'R' and uplo = 'U', ltau = LOCc(n_a),
if side = 'R' and uplo = 'L', ltau = LOCc(ja+n-2).
```

tau[*i*] must contain the scalar factor of the elementary reflector $H(i+1)$, as returned by `p?hetrd` ($0 \leq i < ltau$). *tau* is tied to the distributed matrix *A*.

c

(local)

Pointer into the local memory to an array of size *lld_c***LOCc*(*jc+n-1*). Contains the local pieces of the distributed matrix sub (*C*).

ic, jc

(global) The row and column indices in the global matrix *C* indicating the first row and the first column of the submatrix *C*, respectively.

descc

(global and local) array of size *dlen_*. The array descriptor for the distributed matrix *C*.

work

(local)

Workspace array of size *lwork*.

lwork

(local or global) size of *work*, must be at least:

```
If uplo = 'U',
  iaa= ia; jaa= ja+1, icc= ic; jcc= jc;
else uplo = 'L',
  iaa= ia+1, jaa= ja;
If side = 'L',
  icc= ic+1; jcc= jc;
else icc= ic; jcc= jc+1;
end if
end if

If side = 'L',
  mi= m-1; ni= n
  lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) +
  nb_a*nb_a
else
  If side = 'R',
    mi= m; ni = n-1;
    lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 +
    max(npa0+numroc(numroc(ni+icoffc, nb_a, 0, 0, NPCOL), nb_a,
    0, 0, lcmq), mpc0))*nb_a) + nb_a*nb_a
  end if
  where lcmq = lcm/NPCOL with lcm = ilcm(NPROW, NPCOL),
```

```

iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),
icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

`mod(x,y)` is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`. If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	Overwritten by the product $Q*\text{sub}(C)$, or $Q'*\text{sub}(C)$, or $\text{sub}(C)*Q'$, or $\text{sub}(C)*Q$.
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?stebz

Computes the eigenvalues of a symmetric tridiagonal matrix by bisection.

Syntax

```

void psstebz (MKL_INT *ictxt , char *range , char *order , MKL_INT *n , float *vl ,
float *vu , MKL_INT *il , MKL_INT *iu , float *abstol , float *d , float *e , MKL_INT
*m , MKL_INT *nsplit , float *w , MKL_INT *iblock , MKL_INT *isplit , float *work ,
MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

```

```
void pdstebz (MKL_INT *ictxt , char *range , char *order , MKL_INT *n , double *vl ,
double *vu , MKL_INT *il , MKL_INT *iu , double *abstol , double *d , double *e ,
MKL_INT *m , MKL_INT *nsplit , double *w , MKL_INT *iblock , MKL_INT *isplit , double
*work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `pdstebz` function computes the eigenvalues of a symmetric tridiagonal matrix in parallel. These may be all eigenvalues, all eigenvalues in the interval $[vl, vu]$, or the eigenvalues il through iu . A static partitioning of work is done at the beginning of `pdstebz` which results in all processes finding an (almost) equal number of eigenvalues.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>ictxt</i>	(global) The BLACS context handle.
<i>range</i>	(global) Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the function computes all eigenvalues. If <i>range</i> = 'V', the function computes eigenvalues in the interval $[vl, vu]$. If <i>range</i> = 'I', the function computes eigenvalues il through iu .
<i>order</i>	(global) Must be 'B' or 'E'. If <i>order</i> = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block. If <i>order</i> = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.
<i>n</i>	(global) The order of the tridiagonal matrix T ($n \geq 0$).
<i>vl, vu</i>	(global) If <i>range</i> = 'V', the function computes the lower and the upper bounds for the eigenvalues on the interval $[1, vu]$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	(global) Constraint: $1 \leq il \leq iu \leq n$. If <i>range</i> = 'I', the index of the smallest eigenvalue is returned for il and of the largest eigenvalue for iu (assuming that the eigenvalues are in ascending order) must be returned. If <i>range</i> = 'A' or 'V', il and iu are not referenced.

<i>abstol</i>	<p>(global)</p> <p>The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width <i>abstol</i>. If <i>abstol</i> ≤ 0, then the tolerance is taken as <i>ulp</i> <i>T</i> , where <i>ulp</i> is the machine precision, and <i>T</i> means the 1-norm of <i>T</i>.</p> <p>Eigenvalues will be computed most accurately when <i>abstol</i> is set to the underflow threshold <code>slamch('U')</code>, not 0. Note that if eigenvectors are desired later by inverse iteration (<code>p?stein</code>), <i>abstol</i> should be set to <code>2*p?lamch('S')</code>.</p>
<i>d</i>	<p>(global)</p> <p>Array of size <i>n</i>.</p> <p>Contains <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i>. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than the $\text{overflow}^{(1/2)} * \text{underflow}^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.</p>
<i>e</i>	<p>(global)</p> <p>Array of size <i>n</i> - 1.</p> <p>Contains (<i>n</i>-1) off-diagonal elements of the tridiagonal matrix <i>T</i>. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $\text{overflow}^{(1/2)} * \text{underflow}^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.</p>
<i>work</i>	<p>(local)</p> <p>Array of size <code>max(5<i>n</i>, 7)</code>. This is a workspace array.</p>
<i>lwork</i>	<p>(local) The size of the <i>work</i> array must be <math>\geq \text{max}(5<i>n</i>, 7)</math>.</p> <p>If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pxerbla</code>.</p>
<i>iwork</i>	<p>(local) Array of size <code>max(4<i>n</i>, 14)</code>. This is a workspace array.</p>
<i>liwork</i>	<p>(local) The size of the <i>iwork</i> array must <math>\geq \text{max}(4<i>n</i>, 14, \text{NPROCS})</math>.</p> <p>If <i>liwork</i> = -1, then <i>liwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pxerbla</code>.</p>

Output Parameters

<i>m</i>	(global) The actual number of eigenvalues found. $0 \leq m \leq n$
<i>nsplit</i>	(global) The number of diagonal blocks detected in <i>T</i> . $1 \leq \text{nsplit} \leq n$
<i>w</i>	<p>(global)</p> <p>Array of size <i>n</i>. On exit, the first <i>m</i> elements of <i>w</i> contain the eigenvalues on all processes.</p>

iblock

(global)

Array of size n . At each row/column j where $e[j-1]$ is zero or small, the matrix T is considered to split into a block diagonal matrix. On exit *iblock*[i] specifies which block (from 1 to the number of blocks) the eigenvalue $w[i]$ belongs to.

NOTE

In the (theoretically impossible) event that bisection does not converge for some or all eigenvalues, *info* is set to 1 and the ones for which it did not are identified by a negative block number.

isplit

(global)

Array of size n .

Contains the splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to *isplit*[0], the second of rows/columns *isplit*[0]+1 through *isplit*[1], and so on, and the *nsplit*-th submatrix consists of rows/columns *isplit*[*nsplit*-2]+1 through *isplit*[*nsplit*-1]= n . (Only the first *nsplit* elements are used, but since the *nsplit* values are not known, n words must be reserved for *isplit*.)

info

(global)

If *info* = 0, the execution is successful.

If *info* < 0, if *info* = - i , the i -th argument has an illegal value.

If *info* > 0, some or all of the eigenvalues fail to converge or are not computed.

If *info* = 1, bisection fails to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.

If *info* = 2, mismatch between the number of eigenvalues output and the number desired.

If *info* = 3: *range*='I', and the Gershgorin interval initially used is incorrect. No eigenvalues are computed. Probable cause: the machine has a sloppy floating-point arithmetic. Increase the *fudge* parameter, recompile, and try again.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?stedc

Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix in parallel.

Syntax

```
void psstedc (const char* compz, const MKL_INT* n, float* d, float* e, float* q, const
MKL_INT* iq, const MKL_INT* jq, const MKL_INT* descq, float* work, MKL_INT* lwork,
MKL_INT* iwork, const MKL_INT* liwork, MKL_INT* info);
```

```
void pdstedc (const char* compz, const MKL_INT* n, double* d, double* e, double* q,
const MKL_INT* iq, const MKL_INT* jq, const MKL_INT* descq, double* work, MKL_INT*
lwork, MKL_INT* iwork, const MKL_INT* liwork, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

p?stedc computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix in parallel, using the divide and conquer algorithm.

Input Parameters

<i>compz</i>	<p>= 'N': Compute eigenvalues only. (NOT IMPLEMENTED YET)</p> <p>= 'I': Compute eigenvectors of tridiagonal matrix also.</p> <p>= 'V': Compute eigenvectors of original dense symmetric matrix also. On entry, Z contains the orthogonal matrix used to reduce the original matrix to tridiagonal form. (NOT IMPLEMENTED YET)</p>
<i>n</i>	<p>(global)</p> <p>The order of the tridiagonal matrix T. $n \geq 0$.</p>
<i>d</i>	<p>(global)</p> <p>Array, size (n)</p> <p>On entry, the diagonal elements of the tridiagonal matrix.</p>
<i>e</i>	<p>(global)</p> <p>Array, size ($n-1$).</p> <p>On entry, the subdiagonal elements of the tridiagonal matrix.</p>
<i>iq</i>	<p>(global)</p> <p>Q's global row index, which points to the beginning of the submatrix which is to be operated on.</p>
<i>jq</i>	<p>(global)</p> <p>Q's global column index, which points to the beginning of the submatrix which is to be operated on.</p>
<i>descq</i>	<p>(global and local)</p> <p>Array of size $dlen_$.</p> <p>The array descriptor for the distributed matrix Q.</p>
<i>work</i>	<p>(local)</p> <p>Array, size ($lwork$)</p>
<i>lwork</i>	<p>(local)</p> <p>The size of the array <i>work</i>.</p> <p>$lwork = 6*n + 2*NP*NQ$</p> <p>$NP = \text{numroc}(n, NB, MYROW, DESCQ(rsrc_), NPROW)$</p>

$NQ = \text{numroc}(n, NB, MYCOL, \text{DESCQ}(csrc_), NPCOL)$

`numroc` is a ScaLAPACK tool function.

If `lwork = -1`, the `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum size for the `work` array. The required workspace is returned as the first element of `work` and no error message is issued by `pxerbla`.

`iwork` (local)
Array, size (`liwork`)

`liwork` The size of the array `iwork`.
 $liwork = 2 + 7*n + 8*NPCOL$

Output Parameters

`d` On exit, if `info = 0`, the eigenvalues in descending order.

`q` (local)
Array, local size (`lld_q, LOCc(jq+n-1)`)
`q` contains the orthonormal eigenvectors of the symmetric tridiagonal matrix.
On output, `q` is distributed across the P processes in block cyclic format.

`work` On output, `work[0]` returns the workspace needed.

`iwork` On exit, if `liwork > 0`, `iwork[0]` returns the optimal `liwork`.

`info` (global)
= 0: successful exit.
< 0: If the *i*-th argument is an array and the *j*-th entry had an illegal value, then `info = -(i*100+j)`, if the *i*-th argument is a scalar and had an illegal value, then `info = -i`.
> 0: The algorithm failed to compute the `info/(n+1)`-th eigenvalue while working on the submatrix lying in global rows and columns `mod(info,n+1)`.

psstein

Computes the eigenvectors of a tridiagonal matrix using inverse iteration.

Syntax

```
void psstein (MKL_INT *n , float *d , float *e , MKL_INT *m , float *w , MKL_INT
*iblock , MKL_INT *isplit , float *orfac , float *z , MKL_INT *iz , MKL_INT *jz ,
MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT *info );
```

```

void pdstein (MKL_INT *n , double *d , double *e , MKL_INT *m , double *w , MKL_INT
*iblock , MKL_INT *isplit , double *orfac , double *z , MKL_INT *iz , MKL_INT *jz ,
MKL_INT *descz , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , double *gap , MKL_INT *info );

void pcstein (MKL_INT *n , float *d , float *e , MKL_INT *m , float *w , MKL_INT
*iblock , MKL_INT *isplit , float *orfac , MKL_Complex8 *z , MKL_INT *iz , MKL_INT *jz ,
MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT *info );

void pzstein (MKL_INT *n , double *d , double *e , MKL_INT *m , double *w , MKL_INT
*iblock , MKL_INT *isplit , double *orfac , MKL_Complex16 *z , MKL_INT *iz , MKL_INT
*jz , MKL_INT *descz , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT
*liwork , MKL_INT *ifail , MKL_INT *iclustr , double *gap , MKL_INT *info );

```

Include Files

- mkl_scalapack.h

Description

The `p?stein` function computes the eigenvectors of a symmetric tridiagonal matrix T corresponding to specified eigenvalues, by inverse iteration. `p?stein` does not orthogonalize vectors that are on different processes. The extent of orthogonalization is controlled by the input parameter `lwork`. Eigenvectors that are to be orthogonalized are computed by the same process. `p?stein` decides on the allocation of work among the processes and then calls `?stein2` (modified LAPACK function) on each individual process. If insufficient workspace is allocated, the expected orthogonalization may not be done.

NOTE

If the eigenvectors obtained are not orthogonal, increase `lwork` and run the code again.

$p = \text{NPROW} * \text{NPCOL}$ is the total number of processes.

Input Parameters

n	(global) The order of the matrix T ($n \geq 0$).
m	(global) The number of eigenvectors to be returned.
d, e, w	(global) Arrays: d of size n contains the diagonal elements of T . e of size $n-1$ contains the off-diagonal elements of T . w of size m contains all the eigenvalues grouped by split-off block. The eigenvalues are supplied from smallest to largest within the block. (Here the output array w from <code>p?stebz</code> with order = 'B' is expected. The array should be replicated in all processes.)
$iblock$	(global)

Array of size n . The submatrix indices associated with the corresponding eigenvalues in w : 1 for eigenvalues belonging to the first submatrix from the top, 2 for those belonging to the second submatrix, etc. (The output array $iblock$ from `p?stebz` is expected here).

<code>isplit</code>	<p>(global)</p> <p>Array of size n. The splitting points at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to <code>isplit[0]</code>, the second of rows/columns <code>isplit[0]+1</code> through <code>isplit[1]</code>, and so on, and the $nsplit$-th submatrix consists of rows/columns <code>isplit[nsplit-2]+1</code> through <code>isplit[nsplit-1]=n</code>. (The output array <code>isplit</code> from <code>p?stebz</code> is expected here.)</p>
<code>orfac</code>	<p>(global)</p> <p><code>orfac</code> specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues within $orfac * T$ of each other are to be orthogonalized. However, if the workspace is insufficient (see <code>lwork</code>), this tolerance may be decreased until all eigenvectors can be stored in one process. No orthogonalization is done if <code>orfac</code> is equal to zero. A default value of 1000 is used if <code>orfac</code> is negative. <code>orfac</code> should be identical on all processes</p>
<code>iz, jz</code>	<p>(global) The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z, respectively.</p>
<code>descz</code>	<p>(global and local) array of size <code>dlen_</code>. The array descriptor for the distributed matrix Z.</p>
<code>work</code>	<p>(local).</p> <p>Workspace array of size <code>lwork</code>.</p>
<code>lwork</code>	<p>(local)</p> <p><code>lwork</code> controls the extent of orthogonalization which can be done. The number of eigenvectors for which storage is allocated on each process is $nvec = \text{floor}((lwork - \max(5*n, np00*mq00))/n)$. Eigenvectors corresponding to eigenvalue clusters of size $(nvec - \text{ceil}(m/p) + 1)$ are guaranteed to be orthogonal (the orthogonality is similar to that obtained from <code>?stein2</code>).</p>

NOTE

`lwork` must be no smaller than $\max(5*n, np00*mq00) + \text{ceil}(m/p) * n$ and should have the same input value on all processes.

It is the minimum value of `lwork` input on different processes that is significant.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

<code>iwork</code>	(local)
--------------------	---------

Workspace array of size $3n+p+1$.

liwork

(local) The size of the array *iwork*. It must be greater than $3*n+p+1$.

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

z

(local)

Array of size *descz*[*dlen* - 1], $n/\text{NPCOL} + \text{NB}$). *z* contains the computed eigenvectors associated with the specified eigenvalues. Any vector which fails to converge is set to its current iterate after MAXIT iterations (See [?stein2](#)). On output, *z* is distributed across the *p* processes in block cyclic format.

work

On exit, *work*[0] gives a lower bound on the workspace (*lwork*) that guarantees the user desired orthogonalization (see *orfac*). Note that this may overestimate the minimum workspace needed.

iwork

On exit, *iwork*[0] contains the amount of integer workspace required.

On exit, the *iwork*[1] through *iwork*[*p*+1] indicate the eigenvectors computed by each process. Process *i* computes eigenvectors indexed *iwork*[*i*+1]+1 through *iwork*[*i*+2].

ifail

(global) Array of size *m*. On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after MAXIT iterations (as in [?stein](#)), then *info* > 0 is returned. If $\text{mod}(\text{info}, m+1) > 0$, then for $i=1$ to $\text{mod}(\text{info}, m+1)$, the eigenvector corresponding to the eigenvalue *w*[*ifail*[*i*-1]-1] failed to converge (*w* refers to the array of eigenvalues on output).

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

iclustr

(global) Array of size $2*p$.

This output array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be orthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*I-1) to *iclustr*(2*I), $i = 1$ to $\text{info}/(m+1)$, could not be orthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr* is a zero terminated array: *iclustr*[2*k-1] ≠ 0 and *iclustr*[2*k] = 0 if and only if *k* is the number of clusters.

gap

(global)

This output array contains the gap between eigenvalues whose eigenvectors could not be orthogonalized. The info/m output values in this array correspond to the $\text{info}/(m+1)$ clusters indicated by the

array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as $(O(n) * macheps) / gap[i-1]$.

info

(global)

If *info* = 0, the execution is successful.

If *info* < 0: If the *i*-th argument is an array and the *j*-th entry, indexed *j*-1, had an illegal value, then *info* = $-(i*100+j)$,

If the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

If *info* < 0: if *info* = $-i$, the *i*-th argument had an illegal value.

If *info* > 0: if $\text{mod}(\text{info}, m+1) = i$, then *i* eigenvectors failed to converge in MAXIT iterations. Their indices are stored in the array *ifail*. If $\text{info}/(m+1) = i$, then eigenvectors corresponding to *i* clusters of eigenvalues could not be orthogonalized due to insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Nonsymmetric Eigenvalue Problems: ScaLAPACK Computational Routines

This section describes ScaLAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

To solve a nonsymmetric eigenvalue problem with ScaLAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained.

Table "Computational Routines for Solving Nonsymmetric Eigenproblems" lists ScaLAPACK routines for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$, as well as routines for solving eigenproblems with Hessenberg matrices, and multiplying the matrix after reduction.

Computational Routines for Solving Nonsymmetric Eigenproblems

Operation performed	General matrix	Orthogonal/Unitary matrix	Hessenberg matrix
Reduce to Hessenberg form $A = QHQ^H$	p?gehrd		
Multiply the matrix after reduction		p?ormhr / p?unmhr	
Find eigenvalues and Schur factorization			p?lahqr

[p?gehrd](#)

Reduces a general matrix to upper Hessenberg form.

Syntax

```
void psgehrd (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT
*info );
```

```
void pdgehrd (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT
*info );
```

```
void pcgehrd (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
```

```
void pzgehrd (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The pzgehrd function reduces a real/complex general distributed matrix sub(*A*) to upper Hessenberg form *H* by an orthogonal or unitary similarity transformation

$$Q^* \text{sub}(A) Q = H,$$

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<i>n</i>	(global). The order of the distributed matrix sub(<i>A</i>) ($n \geq 0$).
<i>ilo, ihi</i>	(global). It is assumed that sub(<i>A</i>) is already upper triangular in rows <i>ia:ia+ilo-2</i> and <i>ia+ihi:ia+n-1</i> and columns <i>ja:ja+ilo-2</i> and <i>ja+ihi:ja+n-1</i> . (See <i>Application Notes</i> below). If $n > 0$, $1 \leq ilo \leq ihi \leq n$; otherwise set <i>ilo</i> = 1, <i>ihi</i> = <i>n</i> .
<i>a</i>	(local) Pointer into the local memory to an array of size <i>lld_a*LOCc(ja+n-1)</i> . On entry, this array contains the local pieces of the <i>n</i> -by- <i>n</i> general distributed matrix sub(<i>A</i>) to be reduced.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq NB * NB + NB * \max(ihip+1, ihlp+inlq)$ <p>where $NB = mb_a = nb_a$,</p> $iroffa = \text{mod}(ia-1, NB),$ $icoffa = \text{mod}(ja-1, NB),$ $ioff = \text{mod}(ia+ilo-2, NB), iarow = \text{indxg2p}(ia, NB, MYROW, rsrc_a, NPROW), ihip = \text{numroc}(ihi+iroffa, NB, MYROW, iarow, NPROW),$ $ilrow = \text{indxg2p}(ia+ilo-1, NB, MYROW, rsrc_a, NPROW),$ $ihlp = \text{numroc}(ihi-ilo+ioff+1, NB, MYROW, ilrow, NPROW),$


```
ilcol = indxg2p(ja+ilo-1, NB, MYCOL, csrc_a, NPCOL),
inlq = numroc(n-ilo+ioff+1, NB, MYCOL, ilcol, NPCOL),
```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	On exit, the upper triangle and the first subdiagonal of <code>sub(A)</code> are overwritten with the upper Hessenberg matrix H , and the elements below the first subdiagonal, with the array <code>tau</code> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<code>tau</code>	(local). Array of size at least $\max(ja+n-2)$. The scalar factors of the elementary reflectors (see <i>Application Notes</i> below). Elements <code>ja:ja+ilo-2</code> and <code>ja+ihi:ja+n-2</code> of the global vector <code>tau</code> are set to zero. <code>tau</code> is tied to the distributed matrix A .
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

Application Notes

The matrix Q is represented as a product of $(ihi-ilo)$ elementary reflectors

$$Q = H(ilo) * H(ilo+1) * \dots * H(ihi-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$, $v(i+1) = 1$ and $v(ihi+1:n) = 0$; $v(i+2:ihi)$ is stored on exit in `A(ia+ilo+i:ia+ihi-1, ja+ilo+i-2)`, and τ in `tau[ja+ilo+i-3]`. The contents of A

`(ia:ia+n-1, ja:ja+n-1)` are illustrated by the following example, with $n = 7$, $ilo = 2$ and $ihi = 6$:

on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & a & a & a & a & a \\ & & & a & a & a & a \\ & & & & a & a & a \\ & & & & & a & a \\ & & & & & & a \end{bmatrix}$$

on exit

$$\begin{bmatrix} a & a & a & h & h & h & a \\ & a & h & h & h & h & a \\ & & h & h & h & h & h \\ v2 & h & h & h & h & h & \\ v2 & v3 & h & h & h & h & \\ v2 & v3 & v4 & h & h & h & \\ & & & & & & a \end{bmatrix}$$

where a denotes an element of the original matrix $\text{sub}(A)$, h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(ja+ilo+i-2)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormhr

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

```
void psormhr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *ilo ,
MKL_INT *ihi , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau ,
float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork ,
MKL_INT *info );

void pdormhr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *ilo ,
MKL_INT *ihi , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau ,
double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork ,
MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?ormhr` function overwrites the general real distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
$trans = 'T':$	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix of order nq , with $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$.

Q is defined as the product of $ihi-ilo$ elementary reflectors, as returned by `p?gehrd`.

$Q = H(ilo) H(ilo+1) \dots H(ihi-1)$.

Input Parameters

$side$	(global) = $'L'$: Q or Q^T is applied from the left. = $'R'$: Q or Q^T is applied from the right.
$trans$	(global) = $'N'$, no transpose, Q is applied. = $'T'$, transpose, Q^T is applied.
m	(global) The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
n	(global) The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
ilo, ihi	(global) ilo and ihi must have the same values as in the previous call of <code>p?gehrd</code> . Q is equal to the unit matrix except for the distributed submatrix $Q(ia+ilo:ia+ihi-1, ja+ilo:ja+ihi-1)$. If $side = 'L'$, $1 \leq ilo \leq ihi \leq \max(1, m)$; If $side = 'R'$, $1 \leq ilo \leq ihi \leq \max(1, n)$; ilo and ihi are relative indexes.
a	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja+m-1)$ if $side = 'L'$, and $lld_a * LOCC(ja+n-1)$ if $side = 'R'$. Contains the vectors which define the elementary reflectors, as returned by <code>p?gehrd</code> .
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
tau	(local)

Array of size $LOCc(ja+m-2)$ if $side = 'L'$, and $LOCc(ja+n-2)$ if $side = 'R'$.

$\tau[j]$ contains the scalar factor of the elementary reflector $H(j+1)$ as returned by `p?gehrd` ($0 \leq j < \text{size}(\tau)$). τ is tied to the distributed matrix A .

c

(local)

Pointer into the local memory to an array of size $lld_c * LOCc(jc+n-1)$.

Contains the local pieces of the distributed matrix sub(C).

ic, jc

(global) The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C , respectively.

$desc$

(global and local) array of size $dlen_$. The array descriptor for the distributed matrix C .

$work$

(local)

Workspace array with size $lwork$.

$lwork$

(local or global)

The size of the array $work$.

$lwork$ must be at least $iaa = ia + ilo; jaa = ja + ilo - 1;$

If $side = 'L'$,

$mi = ihi - ilo; ni = n; icc = ic + ilo; jcc = jc; lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$

else if $side = 'R'$,

$mi = m; ni = ihi - ilo; icc = ic; jcc = jc + ilo; lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(ni + iroffa, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) * nb_a) + nb_a * nb_a$

end if

where $lcmq = lcm / NPCOL$ with $lcm = ilcm(NPROW, NPCOL)$,

$iroffa = \text{mod}(iaa - 1, mb_a)$,

$icoffa = \text{mod}(jaa - 1, nb_a)$,

$iarow = \text{indxg2p}(iaa, mb_a, MYROW, rsrc_a, NPROW)$,

$npa0 = \text{numroc}(ni + iroffa, mb_a, MYROW, iarow, NPROW)$,

$iroffc = \text{mod}(icc - 1, mb_c)$, $icoffc = \text{mod}(jcc - 1, nb_c)$,

$icrow = \text{indxg2p}(icc, mb_c, MYROW, rsrc_c, NPROW)$,

$iccol = \text{indxg2p}(jcc, nb_c, MYCOL, csrc_c, NPCOL)$,

$mpc0 = \text{numroc}(mi + iroffc, mb_c, MYROW, icrow, NPROW)$,

$nqc0 = \text{numroc}(ni + icoffc, nb_c, MYCOL, iccol, NPCOL)$,

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	<code>sub(C)</code> is overwritten by $Q*\text{sub}(C)$, or $Q'*\text{sub}(C)$, or $\text{sub}(C)*Q'$, or $\text{sub}(C)*Q$.
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmhr

Multiplies a general matrix by the unitary transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

```
void pcunmhr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *ilo ,
MKL_INT *ihi , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex8 *tau , MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzunmhr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *ilo ,
MKL_INT *ihi , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex16 *tau , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mk1_scalapack.h`

Description

This function overwrites the general complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
<i>trans</i> = 'H':	$Q^H * \text{sub}(C)$	$\text{sub}(C) * Q^H$

where Q is a complex unitary distributed matrix of order nq , with $nq = m$ if *side* = 'L' and $nq = n$ if *side* = 'R'.

Q is defined as the product of *ihi-ilo* elementary reflectors, as returned by `p?gehrd`.

$Q = H(ilo) H(ilo+1) \dots H(ihi-1)$.

Input Parameters

<i>side</i>	(global) = 'L': Q or Q^H is applied from the left. = 'R': Q or Q^H is applied from the right.
<i>trans</i>	(global) = 'N', no transpose, Q is applied. = 'C', conjugate transpose, Q^H is applied.
<i>m</i>	(global) The number of rows in the distributed matrix sub (C) ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix sub (C) ($n \geq 0$).
<i>ilo, ihi</i>	(global) These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to <code>p?gehrd</code> . Q is equal to the unit matrix except in the distributed submatrix $Q(ia+ilo:ia+ihi-1, ja+ilo:ja+ihi-1)$. If <i>side</i> = 'L', then $1 \leq ilo \leq ihi \leq \max(1, m)$. If <i>side</i> = 'R', then $1 \leq ilo \leq ihi \leq \max(1, n)$ <i>ilo</i> and <i>ihi</i> are relative indexes.
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOCc(ja+m-1)$ if <i>side</i> = 'L', and $lld_a * LOCc(ja+n-1)$ if <i>side</i> = 'R'. Contains the vectors which define the elementary reflectors, as returned by <code>p?gehrd</code> .
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) Array of size $LOCc(ja+m-2)$, if <i>side</i> = 'L', and $LOCc(ja+n-2)$ if <i>side</i> = 'R'. $tau[j]$ contains the scalar factor of the elementary reflector $H(j+1)$ as returned by <code>p?gehrd</code> ($0 \leq j < \text{size}(tau)$). <i>tau</i> is tied to the distributed matrix A .

<i>c</i>	(local) Pointer into the local memory to an array of size $lld_c * LOCc(jc+n-1)$. Contains the local pieces of the distributed matrix sub(<i>C</i>).
<i>ic, jc</i>	(global) The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descc</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) Workspace array with size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . $lwork$ must be at least $iaa = ia + ilo; jaa = ja + ilo - 1;$ If <i>side</i> = 'L', $mi = ihi - ilo; ni = n; icc = ic + ilo; jcc = jc;$ $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$ else if <i>side</i> = 'R', $mi = m; ni = ihi - ilo; icc = ic; jcc = jc + ilo; lwork \geq$ $\max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(ni + iroffa, mb_a, MYROW, iarow, NPROW),$ $+ icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq),$ $mpc0)) * nb_a) + nb_a * nb_a$ end if where $lcmq = lcm / NPCOL$ with $lcm = ilcm(NPROW, NPCOL),$ $iroffa = \text{mod}(iaa - 1, mb_a),$ $icoffa = \text{mod}(jaa - 1, nb_a),$ $iarow = \text{indxg2p}(iaa, mb_a, MYROW, rsrc_a, NPROW),$ $npa0 = \text{numroc}(ni + iroffa, mb_a, MYROW, iarow, NPROW),$ $iroffc = \text{mod}(icc - 1, mb_c),$ $icoffc = \text{mod}(jcc - 1, nb_c),$ $icrow = \text{indxg2p}(icc, mb_c, MYROW, rsrc_c, NPROW),$ $iccol = \text{indxg2p}(jcc, nb_c, MYCOL, csrc_c, NPCOL),$ $mpc0 = \text{numroc}(mi + iroffc, mb_c, MYROW, icrow, NPROW),$ $nqc0 = \text{numroc}(ni + icoffc, nb_c, MYCOL, iccol, NPCOL),$

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

ilcm, *indxg2p* and *numroc* are ScaLAPACK tool functions; *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the function *blacs_gridinfo*.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	<code>C</code> is overwritten by $Q^* \text{sub}(C)$ or $Q'^* \text{sub}(C)$ or $\text{sub}(C) * Q'$ or $\text{sub}(C) * Q$.
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lahqr

Computes the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form.

Syntax

```
void pslahqr (MKL_INT *wantt, MKL_INT *wantz, MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi,
float *a, MKL_INT *desca, float *wr, float *wi, MKL_INT *iloz, MKL_INT *ihiz, float *z,
MKL_INT *descz, float *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *ilwork, MKL_INT
*info );

void pdlahqr (MKL_INT *wantt, MKL_INT *wantz, MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi,
double *a, MKL_INT *desca, double *wr, double *wi, MKL_INT *iloz, MKL_INT *ihiz, double
*z, MKL_INT *descz, double *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *ilwork,
MKL_INT *info );

void pclahqr (const MKL_INT *wantt, const MKL_INT *wantz, const MKL_INT *n, const
MKL_INT *ilo, const MKL_INT *ihi, MKL_Complex8 *a, const MKL_INT *desca, MKL_Complex8
*w, const MKL_INT *iloz, const MKL_INT *ihiz, MKL_Complex8 *z, const MKL_INT *descz,
MKL_Complex8 *work, const MKL_INT *lwork, const MKL_INT *iwork, const MKL_INT *ilwork,
MKL_INT *info );

void pzlahqr (const MKL_INT *wantt, const MKL_INT *wantz, const MKL_INT *n, const
MKL_INT *ilo, const MKL_INT *ihi, MKL_Complex16 *a, const MKL_INT *desca, MKL_Complex16
*w, const MKL_INT *iloz, const MKL_INT *ihiz, MKL_Complex16 *z, const MKL_INT *descz,
MKL_Complex16 *work, const MKL_INT *lwork, const MKL_INT *iwork, const MKL_INT *ilwork,
MKL_INT *info );
```

Include Files

- `mkc_sscalapack.h`

Description

This is an auxiliary function used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns `ilo` and `ihi`.

NOTE

These restrictions apply to the use of `p?lahqr`:

- The code requires the distributed block size to be square and at least 6.
- The code requires A and Z to be distributed identically and have identical contexts.
- The matrix A must be in upper Hessenberg form. If elements below the subdiagonal are non-zero, the resulting transformations can be nonsimilar.
- All eigenvalues are distributed to all the nodes.

Input Parameters

<code>wantt</code>	(global) If <code>wantt ≠ 0</code> , the full Schur form <i>T</i> is required; If <code>wantt = 0</code> , only eigenvalues are required.
<code>wantz</code>	(global) If <code>wantz ≠ 0</code> , the matrix of Schur vectors <i>Z</i> is required; If <code>wantz = 0</code> , Schur vectors are not required.
<code>n</code>	(global) The order of the Hessenberg matrix <i>A</i> (and <i>z</i> if <code>wantz</code> is non-zero). $n ≥ 0$.
<code>ilo, ihi</code>	(global) It is assumed that <i>A</i> is already upper quasi-triangular in rows and columns <i>ihi</i> +1: <i>n</i> , and that $A(ilo, ilo-1) = 0$ (unless $ilo = 1$). <code>p?lahqr</code> works primarily with the Hessenberg submatrix in rows and columns <i>ilo</i> to <i>ihi</i> , but applies transformations to all of <i>H</i> if <code>wantt</code> is non-zero. $1 ≤ ilo ≤ \max(1, ihi); ihi ≤ n$.
<code>a</code>	(global) Array, of size $lld_a * LOCc(n)$. On entry, the upper Hessenberg matrix <i>A</i> .
<code>desca</code>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<code>iloz, ihiz</code>	(global) Specify the rows of the matrix <i>Z</i> to which transformations must be applied if <code>wantz</code> is non-zero. $1 ≤ iloz ≤ ilo; ihi ≤ ihiz ≤ n$.
<code>z</code>	(global) Array. If <code>wantz</code> is non-zero, on entry <i>z</i> must contain the current matrix <i>Z</i> of transformations accumulated by <code>pdhseqr</code> . If <code>wantz</code> is zero, <i>z</i> is not referenced.
<code>descz</code>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> .
<code>work</code>	(local) Workspace array with size <i>lwork</i> .
<code>lwork</code>	(local) The size of <i>work</i> . <i>lwork</i> is assumed big enough so that $lwork ≥ 3*n + \max(2*\max(lld_z, lld_a) + 2*LOCc(n), 7*ceil(n/hbl) / lcm(NPROW, NPCOL))$.

If `lwork = -1`, then `work[0]` gets set to the above number and the code returns immediately.

`iwork`

(global and local) array of size `ilwork`. Not referenced and can be NULL pointer.

`ilwork`

(local) This holds some of the `iblk` integer arrays. Not referenced and can be NULL pointer.

Output Parameters

`a`

On exit, if `wantt` is non-zero, `A` is upper quasi-triangular in rows and columns `ilo:ihi`, with any 2-by-2 or larger diagonal blocks not yet in standard form. If `wantt` is zero, the contents of `A` are unspecified on exit.

`work[0]`

On exit `work[0]` contains the minimum value of `lwork` required for optimum performance.

`wr, wi`

(global replicated output)

Arrays of size `n` each. The real and imaginary parts, respectively, of the computed eigenvalues `ilo` to `ihi` are stored in the corresponding elements of `wr` and `wi`. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of `wr` and `wi`, say the i -th and $(i+1)$ -th, with `wi[i-1] > 0` and `wi[i] < 0`. If `wantt` is zero, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in `A`. `A` may be returned with larger diagonal blocks until the next release.

`w`

(global replicated output)

Array of size `n`. The computed eigenvalues `ilo` to `ihi` are stored in the corresponding elements of `w`. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of `w`, say the i -th and $(i+1)$ -th, with `w[i-1] > 0` and `w[i] < 0`. If `wantt` is zero, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in `A`. `A` may be returned with larger diagonal blocks until the next release.

`z`

On exit `z` has been updated; transformations are applied only to the submatrix `Z(ilo:ihiz, ilo:ihi)`.

`info`

(global)

= 0: the execution is successful.

< 0: the parameter number - `info` is incorrect or inconsistent

> 0: `p?lahqr` failed to compute all the eigenvalues `ilo` to `ihi` in a total of $30 * (ihi - ilo + 1)$ iterations; if `info = i`, elements $i+1:ihi$ of `wr` and `wi` contain the eigenvalues that have been successfully computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

`p?trevc`

Computes right and/or left eigenvectors of a complex upper triangular matrix in parallel.

Syntax

```
void pctrvc (const char* side, const char* howmny, const MKL_INT* select, const
MKL_INT* n, MKL_Complex8* t, const MKL_INT* descv, MKL_Complex8* vl, const MKL_INT*
descvl, MKL_Complex8* vr, const MKL_INT* descvr, const MKL_INT* mm, MKL_INT* m,
MKL_Complex8* work, float* rwork, MKL_INT* info);

void pztrvc (const char* side, const char* howmny, const MKL_INT* select, const
MKL_INT* n, MKL_Complex16* t, const MKL_INT* descv, MKL_Complex16* vl, const MKL_INT*
descvl, MKL_Complex16* vr, const MKL_INT* descvr, const MKL_INT* mm, MKL_INT* m,
MKL_Complex16* work, double* rwork, MKL_INT* info);

void pdtrvc (const char* side, const char* howmny, const MKL_INT* select, const
MKL_INT* n, double* t, const MKL_INT* descv, double* vl, const MKL_INT* descvl, double*
vr, const MKL_INT* descvr, const MKL_INT* mm, MKL_INT* m, double* work, MKL_INT* info);

void psttrvc (const char* side, const char* howmny, const MKL_INT* select, const
MKL_INT* n, float* t, const MKL_INT* descv, float* vl, const MKL_INT* descvl, float* vr,
const MKL_INT* descvr, const MKL_INT* mm, MKL_INT* m, float* work, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

`p?trvc` computes some or all of the right and/or left eigenvectors of a complex upper triangular matrix T in parallel.

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w are defined by:

$$T*x = w*x,$$

$$y'*T = w*y'$$

where y' denotes the conjugate transpose of the vector y .

If all eigenvectors are requested, the routine may either return the matrices X and/or Y of right or left eigenvectors of T , or the products $Q*X$ and/or $Q*Y$, where Q is an input unitary matrix. If T was obtained from the Schur factorization of an original matrix $A = Q*T*Q'$, then $Q*X$ and $Q*Y$ are the matrices of right or left eigenvectors of A .

Input Parameters

<i>side</i>	(global) = 'R': compute right eigenvectors only; = 'L': compute left eigenvectors only; = 'B': compute both right and left eigenvectors.
<i>howmny</i>	(global) = 'A': compute all right and/or left eigenvectors; = 'B': compute all right and/or left eigenvectors, and backtransform them using the input matrices supplied in <i>vr</i> and/or <i>vl</i> ; = 'S': compute selected right and/or left eigenvectors, specified by the logical array <i>select</i> .
<i>select</i>	(global)

	<p>Array, size (n)</p> <p>If $howmny = 'S'$, $select$ specifies the eigenvectors to be computed.</p> <p>If $howmny = 'A'$ or $'B'$, $select$ is not referenced. To select the eigenvector corresponding to the j-th eigenvalue, $select[j - 1]$ must be set to non-zero.</p>
n	<p>(global)</p> <p>The order of the matrix T. $n \geq 0$.</p>
t	<p>(local)</p> <p>Array, size $lld_t * LOCC(n)$.</p> <p>The upper triangular matrix T. T is modified, but restored on exit.</p>
$desct$	<p>(global and local)</p> <p>Array of size $dlen_$.</p> <p>The array descriptor for the distributed matrix T.</p>
vl	<p>(local)</p> <p>Array, size $(descvl(lld_), mm)$</p> <p>On entry, if $side = 'L'$ or $'B'$ and $howmny = 'B'$, vl must contain an n-by-n matrix Q (usually the unitary matrix Q of Schur vectors returned by <code>?hseqr</code>).</p>
$descvl$	<p>(global and local)</p> <p>Array of size $dlen_$.</p> <p>The array descriptor for the distributed matrix VL.</p>
vr	<p>(local)</p> <p>Array, size $descvr(lld_)*mm$.</p> <p>On entry, if $side = 'R'$ or $'B'$ and $howmny = 'B'$, vr must contain an n-by-n matrix Q (usually the unitary matrix Q of Schur vectors returned by <code>?hseqr</code>).</p>
$descvr$	<p>(global and local)</p> <p>Array of size $dlen_$.</p> <p>The array descriptor for the distributed matrix VR.</p>
mm	<p>(global)</p> <p>The number of columns in the arrays vl and/or vr. $mm \geq m$.</p>
$work$	<p>(local)</p> <p>Array, size $(2 * desct(lld_))$</p> <p>Additional workspace may be required if <code>p?lattr</code>s is updated to use $work$.</p>
$rwork$	<p>Array, size $(desct(lld_))$</p>

Output Parameters

<i>t</i>	The upper triangular matrix <i>T</i> . <i>T</i> is modified, but restored on exit.
<i>vl</i>	On exit, if <i>side</i> = 'L' or 'B', <i>vl</i> contains: if <i>howmny</i> = 'A', the matrix <i>Y</i> of left eigenvectors of <i>T</i> ; if <i>howmny</i> = 'B', the matrix <i>Q*Y</i> ; if <i>howmny</i> = 'S', the left eigenvectors of <i>T</i> specified by <i>select</i> , stored consecutively in the columns of <i>vl</i> , in the same order as their eigenvalues. If <i>side</i> = 'R', <i>vl</i> is not referenced.
<i>vr</i>	On exit, if <i>side</i> = 'R' or 'B', <i>vr</i> contains: if <i>howmny</i> = 'A', the matrix <i>X</i> of right eigenvectors of <i>T</i> ; if <i>howmny</i> = 'B', the matrix <i>Q*X</i> ; if <i>howmny</i> = 'S', the right eigenvectors of <i>T</i> specified by <i>select</i> , stored consecutively in the columns of <i>vr</i> , in the same order as their eigenvalues. If <i>side</i> = 'L', <i>vr</i> is not referenced.
<i>m</i>	(global) The number of columns in the arrays <i>vl</i> and/or <i>vr</i> actually used to store the eigenvectors. If <i>howmny</i> = 'A' or 'B', <i>m</i> is set to <i>n</i> . Each selected eigenvector occupies one column.
<i>info</i>	(global) = 0: successful exit < 0: if <i>info</i> = -i, the i-th argument had an illegal value

Application Notes

The algorithm used in this program is basically backward (forward) substitution. Scaling should be used to make the code robust against possible overflow. But scaling has not yet been implemented in `p?lattrsv` which is called by this routine to solve the triangular systems. `p?lattrsv` just calls `p?trsv`.

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be $|x| + |y|$.

Singular Value Decomposition: ScaLAPACK Driver Routines

This section describes ScaLAPACK routines for computing the singular value decomposition (SVD) of a general *m*-by-*n* matrix *A* (see LAPACK "Singular Value Decomposition").

To find the SVD of a general matrix *A*, this matrix is first reduced to a bidiagonal matrix *B* by a unitary (orthogonal) transformation, and then SVD of the bidiagonal matrix is computed. Note that the SVD of *B* is computed using the LAPACK routine `?bdsqr`.

Table "Computational Routines for Singular Value Decomposition (SVD)" lists ScaLAPACK computational routines for performing this decomposition.

Computational Routines for Singular Value Decomposition (SVD)

Operation	General matrix	Orthogonal/unitary matrix
Reduce <i>A</i> to a bidiagonal matrix	<code>p?gebrd</code>	
Multiply matrix after reduction		<code>p?ormbr/p?unmbr</code>

p?gebrd*Reduces a general matrix to bidiagonal form.***Syntax**

```

void psgebrd (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *d , float *e , float *tauq , float *taup , float *work , MKL_INT
*lwork , MKL_INT *info );

void pdgebrd (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *d , double *e , double *tauq , double *taup , double *work , MKL_INT
*lwork , MKL_INT *info );

void pcgebrd (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *d , float *e , MKL_Complex8 *tauq , MKL_Complex8 *taup ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzgebrd (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *d , double *e , MKL_Complex16 *tauq , MKL_Complex16 *taup ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );

```

Include Files

- mkl_scalapack.h

Description

The `p?gebrd` function reduces a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form B by an orthogonal/unitary transformation:

$$Q^* \text{sub}(A) P = B.$$

If $m \geq n$, B is upper bidiagonal; if $m < n$, B is lower bidiagonal.

Input Parameters

m	(global) The number of rows in the distributed matrix $\text{sub}(A)$ ($m \geq 0$).
n	(global) The number of columns in the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) Real pointer into the local memory to an array of size $ld_a * LOCC(ja+n-1)$. On entry, this array contains the distributed matrix $\text{sub}(A)$.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
$work$	(local) Workspace array of size $lwork$.
$lwork$	(local or global) size of $work$, must be at least: $lwork \geq nb * (mpa0 + nqa0 + 1) + nqa0$ where $nb = mb_a = nb_a$, $iroffa = \text{mod}(ia-1, nb)$, $icoffa = \text{mod}(ja-1, nb)$,

```

iarow = indxg2p(ia, nb, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m + iroffa, nb, MYROW, iarow, NPROW),
nqa0 = numroc(n + icoffa, nb, MYCOL, iacol, NPCOL),

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters*a*

On exit, if $m \geq n$, the diagonal and the first superdiagonal of $\text{sub}(A)$ are overwritten with the upper bidiagonal matrix B ; the elements below the diagonal, with the array `tauq`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array `taup`, represent the orthogonal matrix P as a product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B ; the elements below the first subdiagonal, with the array `tauq`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array `taup`, represent the orthogonal matrix P as a product of elementary reflectors. See *Application Notes* below.

d

(local)

Array of size `LOCc(ja+min(m,n)-1)` if $m \geq n$ and `LOCr(ia+min(m,n)-1)` otherwise. The distributed diagonal elements of the bidiagonal matrix B : $d[i] = A(i+1, i+1)$, $0 \leq i < \text{size}(d)$.

d is tied to the distributed matrix A .

e

(local)

Array of size `LOCr(ia+min(m,n)-1)` if $m \geq n$; `LOCc(ja+min(m,n)-2)` otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix B :

If $m \geq n$, $e[i] = A(i+1, i+2)$ for $i = 0, 1, \dots, n-2$; if $m < n$, $e[i] = A(i+2, i+1)$ for $i = 0, 1, \dots, m-2$. *e* is tied to the distributed matrix A .

tauq, taup

(local)

Arrays of size `LOCc(ja+min(m,n)-1)` for `tauq` and `LOCr(ia+min(m,n)-1)` for `taup`. Contain the scalar factors of the elementary reflectors that represent the orthogonal/unitary matrices Q and P , respectively. `tauq` and `taup` are tied to the distributed matrix A . See *Application Notes* below.

<i>work</i> [0]	On exit <i>work</i> [0] contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

Application Notes

The matrices *Q* and *P* are represented as products of elementary reflectors:

If $m \geq n$,

$Q = H(1)*H(2)*...*H(n)$, and $P = G(1)*G(2)*...*G(n-1)$.

Each *H*(*i*) and *G*(*i*) has the form:

$H(i) = I - \tau_{uq} * v * v'$ and $G(i) = I - \tau_{up} * u * u'$

where τ_{uq} and τ_{up} are real/complex scalars, and *v* and *u* are real/complex vectors;

$v(1:i-1) = 0$, $v(i) = 1$, and $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$;

$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

τ_{uq} is stored in $\tau_{uq}[ja+i-2]$ and τ_{up} in $\tau_{up}[ia+i-2]$.

If $m < n$,

$Q = H(1)*H(2)*...*H(m-1)$, and $P = G(1)*G(2)*...*G(m)$

Each *H*(*i*) and *G*(*i*) has the form:

$H(i) = I - \tau_{uq} * v * v'$ and $G(i) = I - \tau_{up} * u * u'$

here τ_{uq} and τ_{up} are real/complex scalars, and *v* and *u* are real/complex vectors;

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

τ_{uq} is stored in $\tau_{uq}[ja+i-2]$ and τ_{up} in $\tau_{up}[ia+i-2]$.

The contents of sub(*A*) on exit are illustrated by the following examples:

$m = 6$ and $n = 5$ ($m > n$):

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5$ and $n = 6$ ($m < n$):

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of B , vi denotes an element of the vector defining $H(i)$, and ui an element of the vector defining $G(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormbr

Multiplies a general matrix by one of the orthogonal matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

```
void psormbr (char *vect , char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT
*k , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT
*info );
```

```
void pdormbr (char *vect , char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT
*k , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- mkl_scalapack.h

Description

If $vect = 'Q'$, the `p?ormbr` function overwrites the general real distributed m -by- n matrix $sub(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q \ sub(C)$	$sub(C) \ Q$
$trans = 'T':$	$Q^T \ sub(C)$	$sub(C) \ Q^T$

If $vect = 'P'$, the function overwrites $sub(C)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$P \ sub(C)$	$sub(C) \ P$
$trans = 'T':$	$P^T \ sub(C)$	$sub(C) \ P^T$

Here Q and P^T are the orthogonal distributed matrices determined by `p?gebrd` when reducing a real distributed matrix $A(ia:*, ja:*)$ to bidiagonal form: $A(ia:*, ja:*) = Q*B*P^T$. Q and P^T are defined as products of elementary reflectors $H(i)$ and $G(i)$ respectively.

Let $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$. Therefore nq is the order of the orthogonal matrix Q or P^T that is applied.

If $vect = 'Q'$, $A(ia:*, ja:*)$ is assumed to have been an nq -by- k matrix:

If $nq \geq k$, $Q = H(1) H(2) \dots H(k)$;

If $nq < k$, $Q = H(1) H(2) \dots H(nq-1)$.

If $vect = 'P'$, $A(ia:*, ja:*)$ is assumed to have been a k -by- nq matrix:

If $k < nq$, $P = G(1) G(2) \dots G(k)$;

If $k \geq nq$, $P = G(1) G(2) \dots G(nq-1)$.

Input Parameters

<i>vect</i>	(global) If <i>vect</i> = 'Q', then Q or Q^T is applied. If <i>vect</i> = 'P', then P or P^T is applied.
<i>side</i>	(global) If <i>side</i> = 'L', then Q or Q^T , P or P^T is applied from the left. If <i>side</i> = 'R', then Q or Q^T , P or P^T is applied from the right.
<i>trans</i>	(global) If <i>trans</i> = 'N', no transpose, Q or P is applied. If <i>trans</i> = 'T', then Q^T or P^T is applied.
<i>m</i>	(global) The number of rows in the distributed matrix sub (C).
<i>n</i>	(global) The number of columns in the distributed matrix sub (C).
<i>k</i>	(global) If <i>vect</i> = 'Q', the number of columns in the original distributed matrix reduced by p?gebrd; If <i>vect</i> = 'P', the number of rows in the original distributed matrix reduced by p?gebrd. Constraints: $k \geq 0$.
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja + \min(nq, k) - 1)$ if <i>vect</i> = 'Q', and $lld_a * LOCC(ja + nq - 1)$ if <i>vect</i> = 'P'. $nq = m$ if <i>side</i> = 'L', and $nq = n$ otherwise. The vectors that define the elementary reflectors $H(i)$ and $G(i)$, whose products determine the matrices Q and P , as returned by p?gebrd. If <i>vect</i> = 'Q', $lld_a \geq \max(1, LOCC(ia + nq - 1))$; If <i>vect</i> = 'P', $lld_a \geq \max(1, LOCC(ia + \min(nq, k) - 1))$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.
<i>tau</i>	(local) Array of size $LOCc(ja+\min(nq, k)-1)$, if <i>vect</i> = 'Q', and $LOCr(ia+\min(nq, k)-1)$, if <i>vect</i> = 'P'. <i>tau</i> [<i>i</i>] must contain the scalar factor of the elementary reflector $H(i+1)$ or $G(i+1)$ which determines <i>Q</i> or <i>P</i> , as returned by <i>pdgebrd</i> in its array argument <i>tauq</i> or <i>taup</i> . <i>tau</i> is tied to the distributed matrix A.
<i>c</i>	(local) Pointer into the local memory to an array of size $lld_c*LOCc(jc+n-1)$. Contains the local pieces of the distributed matrix sub (C).
<i>ic, jc</i>	(global) The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C, respectively.
<i>descc</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix C.
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least: If <i>side</i> = 'L' $nq = m;$ if ((<i>vect</i> = 'Q' and $nq \geq k$) or (<i>vect</i> is not equal to 'Q' and $nq > k$)), $iaa=ia; jaa=ja; mi=m; ni=n; icc=ic; jcc=jc;$ else $iaa= ia+1; jaa=ja; mi=m-1; ni=n; icc=ic+1; jcc= jc;$ end if else If <i>side</i> = 'R', $nq = n;$ if ((<i>vect</i> = 'Q' and $nq \geq k$) or (<i>vect</i> is not equal to 'Q' and $nq > k$)), $iaa=ia; jaa=ja; mi=m; ni=n; icc=ic; jcc=jc;$ else $iaa= ia; jaa= ja+1; mi= m; ni= n-1; icc= ic; jcc= jc+1;$ end if end if If <i>vect</i> = 'Q', If <i>side</i> = 'L', $lwork \geq \max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) + nb_a * nb_a$

```

else if side = 'R',
    lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 + max(npa0 +
numroc(numroc(ni+icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0,
lcmq), mpc0))*nb_a) + nb_a*nb_a
end if
else if vect is not equal to 'Q', if side = 'L',
    lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + max(mqa0 +
numroc(numroc(mi+iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0,
lcmp), nqc0))*mb_a) + mb_a*mb_a
else if side = 'R',
    lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a) + mb_a*mb_a
end if
end if
where lcmp = lcm/NPROW, lcmq = lcm/NPCOL, with lcm =
ilcm(NPROW, NPCOL),
iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(jaa, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(mi+icoffa, nb_a, MYCOL, iacol, NPCOL),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),
icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

mod(*x*, *y*) is the integer remainder of *x*/*y*.

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxxerbla.

Output Parameters

<code>c</code>	On exit, if <code>vect='Q'</code> , <code>sub(C)</code> is overwritten by $Q*\text{sub}(C)$, or $Q'*\text{sub}(C)$, or $\text{sub}(C)*Q'$, or $\text{sub}(C)*Q$; if <code>vect='P'</code> , <code>sub(C)</code> is overwritten by $P*\text{sub}(C)$, or $P'*\text{sub}(C)$, or $\text{sub}(C)*P$, or $\text{sub}(C)*P'$.
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info</code> = $-(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?unmbr

Multiplies a general matrix by one of the unitary transformation matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

```
void pcunmbr (char *vect , char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT
*k , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
```

```
void pzunmbr (char *vect , char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT
*k , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16
*tau , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16
*work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

If `vect = 'Q'`, the `p?unmbr` function overwrites the general complex distributed m -by- n matrix `sub(C) = C(ic:ic+m-1,jc:jc+n-1)` with

	<code>side = 'L'</code>	<code>side = 'R'</code>
<code>trans = 'N':</code>	$Q*\text{sub}(C)$	$\text{sub}(C)*Q$
<code>trans = 'C':</code>	$Q^H*\text{sub}(C)$	$\text{sub}(C)*Q^H$

If `vect = 'P'`, the function overwrites `sub(C)` with

	<code>side = 'L'</code>	<code>side = 'R'</code>
<code>trans = 'N':</code>	$P*\text{sub}(C)$	$\text{sub}(C)*P$
<code>trans = 'C':</code>	$P^H*\text{sub}(C)$	$\text{sub}(C)*P^H$

Here Q and P^H are the unitary distributed matrices determined by `p?gebrd` when reducing a complex distributed matrix $A(ia:*, ja:*)$ to bidiagonal form: $A(ia:*, ja:*) = Q*B*P^H$.

Q and P^H are defined as products of elementary reflectors $H(i)$ and $G(i)$ respectively.

Let $nq = m$ if $side = 'L'$ and $nq = n$ if $side = 'R'$. Therefore nq is the order of the unitary matrix Q or P^H that is applied.

If $vect = 'Q'$, $A(ia:*, ja:*)$ is assumed to have been an nq -by- k matrix:

If $nq \geq k$, $Q = H(1) H(2) \dots H(k)$;

If $nq < k$, $Q = H(1) H(2) \dots H(nq-1)$.

If $vect = 'P'$, $A(ia:*, ja:*)$ is assumed to have been a k -by- nq matrix:

If $k < nq$, $P = G(1) G(2) \dots G(k)$;

If $k \geq nq$, $P = G(1) G(2) \dots G(nq-1)$.

Input Parameters

<i>vect</i>	(global) If <i>vect</i> = 'Q', then Q or Q^H is applied. If <i>vect</i> = 'P', then P or P^H is applied.
<i>side</i>	(global) If <i>side</i> = 'L', then Q or Q^H , P or P^H is applied from the left. If <i>side</i> = 'R', then Q or Q^H , P or P^H is applied from the right.
<i>trans</i>	(global) If <i>trans</i> = 'N', no transpose, Q or P is applied. If <i>trans</i> = 'C', conjugate transpose, Q^H or P^H is applied.
<i>m</i>	(global) The number of rows in the distributed matrix sub (C) $m \geq 0$.
<i>n</i>	(global) The number of columns in the distributed matrix sub (C) $n \geq 0$.
<i>k</i>	(global) If <i>vect</i> = 'Q', the number of columns in the original distributed matrix reduced by <code>p?gebrd</code> ; If <i>vect</i> = 'P', the number of rows in the original distributed matrix reduced by <code>p?gebrd</code> . Constraints: $k \geq 0$.
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja + \min(nq, k) - 1)$ if <i>vect</i> = 'Q', and $lld_a * LOCC(ja + nq - 1)$ if <i>vect</i> = 'P'. $nq = m$ if <i>side</i> = 'L', and $nq = n$ otherwise. The vectors that define the elementary reflectors $H(i)$ and $G(i)$, whose products determine the matrices Q and P , as returned by <code>p?gebrd</code> . If <i>vect</i> = 'Q', $lld_a \geq \max(1, LOCr(ia + nq - 1))$; If <i>vect</i> = 'P', $lld_a \geq \max(1, LOCr(ia + \min(nq, k) - 1))$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A.
<i>tau</i>	(local) Array of size $LOCc(ja + \min(nq, k) - 1)$, if <i>vect</i> = 'Q', and $LOCr(ia + \min(nq, k) - 1)$, if <i>vect</i> = 'P'. <i>tau</i> [<i>i</i>] must contain the scalar factor of the elementary reflector $H(i+1)$ or $G(i+1)$, which determines <i>Q</i> or <i>P</i> , as returned by <i>p?gebrd</i> in its array argument <i>tauq</i> or <i>taup</i> . <i>tau</i> is tied to the distributed matrix A.
<i>c</i>	(local) Pointer into the local memory to an array of size $lld_c * LOCc(jc + n - 1)$. Contains the local pieces of the distributed matrix sub (C).
<i>ic, jc</i>	(global) The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C, respectively.
<i>descc</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix C.
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) size of <i>work</i> , must be at least: If <i>side</i> = 'L' $nq = m;$ if ((<i>vect</i> = 'Q' and $nq \geq k$) or (<i>vect</i> is not equal to 'Q' and $nq > k$)), <i>iaa</i> = <i>ia</i> ; <i>jaa</i> = <i>ja</i> ; <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> ; else <i>iaa</i> = <i>ia</i> +1; <i>jaa</i> = <i>ja</i> ; <i>mi</i> = <i>m</i> -1; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> +1; <i>jcc</i> = <i>jc</i> ; end if else If <i>side</i> = 'R', $nq = n;$ if ((<i>vect</i> = 'Q' and $nq \geq k$) or (<i>vect</i> is not equal to 'Q' and $nq \geq k$)), <i>iaa</i> = <i>ia</i> ; <i>jaa</i> = <i>ja</i> ; <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> ; else <i>iaa</i> = <i>ia</i> ; <i>jaa</i> = <i>ja</i> +1; <i>mi</i> = <i>m</i> ; <i>ni</i> = <i>n</i> -1; <i>icc</i> = <i>ic</i> ; <i>jcc</i> = <i>jc</i> +1; end if end if If <i>vect</i> = 'Q', If <i>side</i> = 'L', $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$ else if <i>side</i> = 'R',

```

lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 +
max(npa0+numroc(numroc(ni+icoffc, nb_a, 0, 0, NPCOL), nb_a,
0, 0, lcmq), mpc0))*nb_a) + nb_a*nb_a
end if
else if vect is not equal to 'Q',
if side = 'L',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 +
max(mqa0+numroc(numroc(mi+iroffc, mb_a, 0, 0, NPROW), mb_a,
0, 0, lcmp), nqc0))*mb_a) + mb_a*mb_a
else if side = 'R',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a) +
mb_a*mb_a
end if
end if
where lcmp = lcm/NPROW, lcmq = lcm/NPCOL, with lcm =
ilcm(NPROW, NPCOL),
iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(jaa, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(mi+icoffa, nb_a, MYCOL, iacol, NPCOL),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),
icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),

```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If `lwork` = -1, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	On exit, if <code>vect='Q'</code> , <code>sub(C)</code> is overwritten by $Q \cdot \text{sub}(C)$, or $Q' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q'$, or $\text{sub}(C) \cdot Q$; if <code>vect='P'</code> , <code>sub(C)</code> is overwritten by $P \cdot \text{sub}(C)$, or $P' \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot P$, or $\text{sub}(C) \cdot P'$.
<code>work[0]</code>	On exit <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> - 1, had an illegal value, then <code>info</code> = $-(i \cdot 100 + j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Generalized Symmetric-Definite Eigenvalue Problems: ScaLAPACK Computational Routines

This section describes ScaLAPACK routines that allow you to reduce the *generalized symmetric-definite eigenvalue problems* (see [LAPACK Generalized Symmetric-Definite Eigenvalue Problems](#)) to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling ScaLAPACK routines (see [Symmetric Eigenproblems](#)).

Table "Computational Routines for Reducing Generalized Eigenproblems to Standard Problems" lists these routines.

Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to standard problems	<code>p?sygst</code>	<code>p?hegst</code>

p?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

```
void pssygst (MKL_INT *ibtype , char *uplo , MKL_INT *n , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
float *scale , MKL_INT *info );

void pdsygst (MKL_INT *ibtype , char *uplo , MKL_INT *n , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
double *scale , MKL_INT *info );
```

Include Files

- `mk1_scalapack.h`

Description

The `p?sygst` function reduces real symmetric-definite generalized eigenproblems to the standard form.

In the following `sub(A)` denotes $A(ia:ia+n-1, ja:ja+n-1)$ and `sub(B)` denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If `ibtype` = 1, the problem is

$$\text{sub}(A) \cdot x = \lambda \cdot \text{sub}(B) \cdot x,$$

and $\text{sub}(A)$ is overwritten by $\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$.

If $\text{ibtype} = 2$ or 3 , the problem is

$\text{sub}(A) * \text{sub}(B) * x = \lambda * x$, or $\text{sub}(B) * \text{sub}(A) * x = \lambda * x$,

and $\text{sub}(A)$ is overwritten by $U * \text{sub}(A) * U^T$, or $L^T * \text{sub}(A) * L$.

$\text{sub}(B)$ must have been previously factorized as $U^T * U$ or $L * L^T$ by `p?potrf`.

Input Parameters

<i>ibtype</i>	(global) Must be 1 or 2 or 3. If $\text{itype} = 1$, compute $\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$; If $\text{itype} = 2$ or 3 , compute $U * \text{sub}(A) * U^T$, or $L^T * \text{sub}(A) * L$.
<i>uplo</i>	(global) Must be 'U' or 'L'. If $\text{uplo} = \text{'U'}$, the upper triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as $U^T * U$. If $\text{uplo} = \text{'L'}$, the lower triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as $L * L^T$.
<i>n</i>	(global) The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size $\text{ld}_a * \text{LOCc}(j_a + n - 1)$. On entry, the array contains the local pieces of the n -by- n symmetric distributed matrix $\text{sub}(A)$. If $\text{uplo} = \text{'U'}$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If $\text{uplo} = \text{'L'}$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size $\text{dlen}_$. The array descriptor for the distributed matrix A .
<i>b</i>	(local) Pointer into the local memory to an array of size $\text{ld}_b * \text{LOCc}(j_b + n - 1)$. On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of $\text{sub}(B)$ as returned by <code>p?potrf</code> .
<i>ib, jb</i>	(global) The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B , respectively.
<i>descb</i>	(global and local) array of size $\text{dlen}_$. The array descriptor for the distributed matrix B .

Output Parameters

<code>a</code>	On exit, if <code>info = 0</code> , the transformed matrix, stored in the same format as <code>sub(A)</code> .
<code>scale</code>	(global) Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this function. At present, <code>scale</code> is always returned as 1.0, it is returned here to allow for future enhancement.
<code>info</code>	(global) If <code>info = 0</code> , the execution is successful. If <code>info < 0</code> , if the i -th argument is an array and the j -th entry, indexed $j - 1$, had an illegal value, then <code>info = -(i*100+j)</code> ; if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?hegst

Reduces a Hermitian positive-definite generalized eigenvalue problem to the standard form.

Syntax

```
void pchegst (MKL_INT *ibtype , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , float *scale , MKL_INT *info );

void pzhegst (MKL_INT *ibtype , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , double *scale , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?hegst` function reduces complex Hermitian positive-definite generalized eigenproblems to the standard form.

In the following `sub(A)` denotes $A(ia:ia+n-1, ja:ja+n-1)$ and `sub(B)` denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If `ibtype = 1`, the problem is

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x,$$

and `sub(A)` is overwritten by $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$.

If `ibtype = 2` or `3`, the problem is

$$\text{sub}(A) * \text{sub}(B) * x = \lambda * x, \text{ or } \text{sub}(B) * \text{sub}(A) * x = \lambda * x,$$

and `sub(A)` is overwritten by $U * \text{sub}(A) * U^H$, or $L^H * \text{sub}(A) * L$.

`sub(B)` must have been previously factorized as $U^H * U$ or $L * L^H$ by `p?potrf`.

Input Parameters

<i>ibtype</i>	(global) Must be 1 or 2 or 3. If <i>itype</i> = 1, compute $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$; If <i>itype</i> = 2 or 3, compute $U * \text{sub}(A) * U^H$, or $L^H * \text{sub}(A) * L$.
<i>uplo</i>	(global) Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of sub(<i>A</i>) is stored and sub (<i>B</i>) is factored as $U^H * U$. If <i>uplo</i> = 'L', the lower triangle of sub(<i>A</i>) is stored and sub (<i>B</i>) is factored as $L * L^H$.
<i>n</i>	(global) The order of the matrices sub (<i>A</i>) and sub (<i>B</i>) ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size $\text{lld_a} * \text{LOCc}(ja+n-1)$. On entry, the array contains the local pieces of the <i>n</i> -by- <i>n</i> Hermitian distributed matrix sub(<i>A</i>). If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of sub(<i>A</i>) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) Pointer into the local memory to an array of size $\text{lld_b} * \text{LOCc}(jb+n-1)$. On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of sub (<i>B</i>) as returned by <code>p?potrf</code> .
<i>ib, jb</i>	(global) The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, the transformed matrix, stored in the same format as sub(<i>A</i>).
<i>scale</i>	(global) Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this function. At present, <i>scale</i> is always returned as 1.0, it is returned here to allow for future enhancement.
<i>info</i>	(global)

If *info* = 0, the execution is successful. If *info* < 0, if the *i*-th argument is an array and the *j*-th entry, indexed *j* - 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

ScaLAPACK Driver Routines

Table "ScaLAPACK Driver Routines" lists ScaLAPACK driver routines available for solving systems of linear equations, linear least-squares problems, standard eigenvalue and singular value problems, and generalized symmetric definite eigenproblems.

ScaLAPACK Driver Routines

Type of Problem	Matrix type, storage scheme	Driver
Linear equations	general (partial pivoting)	p?gesv (simple driver) / p?gesvx (expert driver)
	general band (partial pivoting)	p?gbsv (simple driver)
	general band (no pivoting)	p?dbsv (simple driver)
	general tridiagonal (no pivoting)	p?dtsv (simple driver)
	symmetric/Hermitian positive-definite	p?posv (simple driver) / p?posvx (expert driver)
	symmetric/Hermitian positive-definite, band	p?pbsv (simple driver)
	symmetric/Hermitian positive-definite, tridiagonal	p?ptsv (simple driver)
Linear least squares problem	general <i>m</i> -by- <i>n</i>	p?gels
Non-symmetric eigenvalue problem	general	p?geevx (expert driver)
Symmetric eigenvalue problem	symmetric/Hermitian	p?syev / p?heev (simple driver); p?syevd / p?heevd (simple driver with a divide and conquer algorithm); p?syevx / p?heevx (expert driver); p?syevr / p?heevr (simple driver with MRRR algorithm)
Singular value decomposition	general <i>m</i> -by- <i>n</i>	p?gesvd
Generalized symmetric definite eigenvalue problem	symmetric/Hermitian, one matrix also positive-definite	p?sygvx / p?hegvx (expert driver)

[p?geevx](#)

*Computes for an *n*-by-*n* real/complex non-symmetric matrix *A*, the eigenvalues and, optionally, the left and/or right eigenvectors.*

Syntax

```
void psgeevx (const char *balanc, const char *jobvl, const char *jobvr, const char
*sense, const MKL_INT *n, float *a, const MKL_INT *desca, float *wr, float *wi, float
*vl, const MKL_INT *descvl, float *vr, const MKL_INT *descvr, MKL_INT *ilo, MKL_INT
*ihi, float *scale, float *abnrm, float *rconde, float *rcondv, float *work, const
MKL_INT *lwork, MKL_INT *info);
```

```

void pdgeevx (const char *balanc, const char *jobvl, const char *jobvr, const char
*sense, const MKL_INT *n, double *a, const MKL_INT *desca, double *wr, double *wi,
double *vl, const MKL_INT *descvl, double *vr, const MKL_INT *descvr, MKL_INT *ilo,
MKL_INT *ihi, double *scale, double *abnrm, double *rconde, double *rcondv, double
*work, const MKL_INT *lwork, MKL_INT *info);

void pcgeevx (const char *balanc, const char *jobvl, const char *jobvr, const char
*sense, const MKL_INT *n, MKL_Complex8 *a, const MKL_INT *desca, MKL_Complex8 *w,
MKL_Complex8 *vl, const MKL_INT *descvl, MKL_Complex8 *vr, const MKL_INT *descvr,
MKL_INT *ilo, MKL_INT *ihi, float *scale, float *abnrm, float *rconde, float *rcondv,
MKL_Complex8 *work, const MKL_INT *lwork, MKL_INT *info);

void pzgeevx (const char *balanc, const char *jobvl, const char *jobvr, const char
*sense, const MKL_INT *n, MKL_Complex16 *a, const MKL_INT *desca, MKL_Complex16 *w,
MKL_Complex16 *vl, const MKL_INT *descvl, MKL_Complex16 *vr, const MKL_INT *descvr,
MKL_INT *ilo, MKL_INT *ihi, double *scale, double *abnrm, double *rconde, double
*rcondv, MKL_Complex16 *work, const MKL_INT *lwork, MKL_INT *info);

```

Include Files

- mkl_scalapack.h

Description

The `p?geevx` function computes for an n -by- n real/complex non-symmetric matrix A , the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (*ilo*, *ihi*, *scale*, and *abnrm*), reciprocal condition numbers for the eigenvalues (*rconde*).

The right eigenvector v of A satisfies

$$A \cdot v = \lambda \cdot v$$

where λ is its eigenvalue.

The left eigenvector u of A satisfies.

$$u^H A = \lambda u^H$$

where u^H denotes the conjugate transpose of u . The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation $D \cdot A \cdot \text{inv}(D)$, where D is a diagonal matrix, to make its rows and columns closer in norm and the condition number of its eigenvalues smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers in exact arithmetic, but diagonal scaling will.

NOTE

The current version doesn't support computation of the reciprocal condition numbers for the right eigenvectors.

Current Notes and Restrictions

All the `p?geevx` interfaces call `p?lahqr` for computing eigenvalues and eigenvectors of the Hessenberg matrices. There are several restrictions for the usage of `p?lahqr`, which include:

- The current implementation of `p?lahqr` requires the distributed block size to be square and at least six (6); unlike simpler codes like LU, this algorithm is extremely sensitive to block size.

- The current implementation of `p?lahqr` requires that input matrix A , the left and right eigenvector matrices VR and/or VL to be distributed identically and have identical context.

Parameters

<i>balanc</i>	<p>(global). Must be 'N', 'P', 'S', or 'B'. Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues.</p> <p>If <i>balanc</i> = 'N', do not diagonally scale or permute;</p> <p>If <i>balanc</i> = 'P', perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale;</p> <p>If <i>balanc</i> = 'S', diagonally scale the matrix, that is, replace A by $D*A*inv(D)$, where D is a diagonal matrix chosen to make the rows and columns of A more equal in norm. Do not permute;</p> <p>If <i>balanc</i> = 'B', both diagonally scale and permute A.</p> <p>Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.</p>
<i>jobvl</i>	<p>(global). Must be 'N' or 'V'.</p> <p>If <i>jobvl</i> = 'N', left eigenvectors of A are not computed;</p> <p>If <i>jobvl</i> = 'V', left eigenvectors of A are computed.</p> <p>If <i>sense</i> = 'E', then <i>jobvl</i> must be 'V'.</p>
<i>jobvr</i>	<p>(global). Must be 'N' or 'V'.</p> <p>If <i>jobvr</i> = 'N', right eigenvectors of A are not computed;</p> <p>If <i>jobvr</i> = 'V', right eigenvectors of A are computed.</p> <p>If <i>sense</i> = 'E', then <i>jobvr</i> must be 'V'.</p>
<i>sense</i>	<p>(global). Must be 'N' or 'E'. Determines which reciprocal condition numbers are computed.</p> <p>If <i>sense</i> = 'N', none are computed.</p> <p>If <i>sense</i> = 'E', computed for eigenvalues only.</p>
<i>n</i>	(global) The order of the distributed matrix A ($n \geq 0$).
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a*LOCc(n)$. On entry, this array contains the local pieces of the n-by-n general distributed matrix A to be reduced.</p>
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>wr, wi</i>	(global output) Arrays, size at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

<i>w</i>	(global output) Array, size at least $\max(1, n)$. Contains the computed eigenvalues.
<i>vl</i>	(local output) Pointer into the local memory to an array of size $(DESCVL(LLD_), LOCc(n))$. If <i>jobvl</i> = 'N', <i>vl</i> is not referenced. If <i>jobvl</i> = 'V', the <i>vl</i> parameter contains the local pieces of the left eigenvectors of the matrix <i>A</i> .
<i>descvl</i>	(global and local input) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>vl</i> .
<i>vr</i>	(local output) Pointer into the local memory to an array of size $(DESCVR(LLD_), LOCc(n))$. If <i>jobvr</i> = 'N', <i>vr</i> is not referenced. If <i>jobvr</i> = 'V', the <i>vr</i> parameter contains the local pieces of the right eigenvectors of the matrix <i>A</i> .
<i>descvr</i>	(global and local input) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>vr</i> .
<i>ilo, ihi</i>	(global output) <i>ilo</i> and <i>ihi</i> are integer values determined when <i>A</i> was balanced. The balanced $A(i, j) = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$. If <i>balanc</i> = 'N' or 'S', <i>ilo</i> = 1 and <i>ihi</i> = <i>n</i> .
<i>scale</i>	(global output) Array, size at least $\max(1, n)$. Details of the permutations and scaling factors applied when balancing <i>A</i> . If $P[j-1]$ is the index of the row and column interchanged with row and column <i>j</i> , and $D[j-1]$ is the scaling factor applied to row and column <i>j</i> , then $scale[j-1] = P[j-1]$, for $j = 1, \dots, ilo-1$ $= D[j-1]$, for $j = ilo, \dots, ihi$ $= P[j-1]$ for $j = ihi+1, \dots, n$. The order in which the interchanges are made is <i>n</i> to <i>ihi</i> +1, then 1 to <i>ilo</i> -1.
<i>abnrm</i>	The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).
<i>rconde</i>	Array, size at least $\max(1, n)$. $rconde[j-1]$ is the reciprocal condition number of the <i>j</i> -th eigenvalue.
<i>rcondv</i>	Not supported in the current version. It could be null pointer.
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) size of the array <i>work</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the minimum size for the work array. These values are returned in the first entry of the <i>work</i> array, and no error message is issued by pxerbla .

info (global)

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-th entry, indexed *j*- 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gesv

Computes the solution to the system of linear equations with a square distributed matrix and multiple right-hand sides.

Syntax

```
void psgesv (MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT
*info );
```

```
void pdgesv (MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
```

```
void pcgesv (MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
```

```
void pzgesv (MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?gesv function computes the solution to a real or complex system of linear equations $\text{sub}(A) * X = \text{sub}(B)$, where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is an *n*-by-*n* distributed matrix and *X* and $\text{sub}(B) = B(ib:ib+n-1, jb:jb+nrhs-1)$ are *n*-by-*nrhs* distributed matrices.

The *LU* decomposition with partial pivoting and row interchanges is used to factor $\text{sub}(A)$ as $\text{sub}(A) = P * L * U$, where *P* is a permutation matrix, *L* is unit lower triangular, and *U* is upper triangular. *L* and *U* are stored in $\text{sub}(A)$. The factored form of $\text{sub}(A)$ is then used to solve the system of equations $\text{sub}(A) * X = \text{sub}(B)$.

Input Parameters

n (global) The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).

nrhs (global) The number of right hand sides, that is, the number of columns of the distributed submatrices *B* and *X* ($nrhs \geq 0$).

a, b (local)

Pointers into the local memory to arrays of local size *a*: $lld_a * LOCC(ja + n - 1)$ and *b*: $lld_b * LOCC(jb + nrhs - 1)$, respectively.

On entry, the array *a* contains the local pieces of the *n*-by-*n* distributed matrix sub(*A*) to be factored.

On entry, the array *b* contains the right hand side distributed matrix sub(*B*).

ia, ja (global) The row and column indices in the global matrix *A* indicating the first row and the first column of sub(*A*), respectively.

desca (global and local) array of size *dlen_*. The array descriptor for the distributed matrix *A*.

ib, jb (global) The row and column indices in the global matrix *B* indicating the first row and the first column of sub(*B*), respectively.

descb (global and local) array of size *dlen_*. The array descriptor for the distributed matrix *B*.

Output Parameters

a Overwritten by the factors *L* and *U* from the factorization $\text{sub}(A) = P * L * U$; the unit diagonal elements of *L* are not stored .

b Overwritten by the solution distributed matrix *X*.

ipiv (local) Array of size $LOCr(m_a) + mb_a$. This array contains the pivoting information. The (local) row *i* of the matrix was interchanged with the (global) row *ipiv*[*i* - 1].

This array is tied to the distributed matrix *A*.

info (global) If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:

If *info* = *k*, *U*(*ia*+*k*-1,*ja*+*k*-1) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gesvx

Uses the LU factorization to compute the solution to the system of linear equations with a square matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```
void psgesvx (char *fact , char *trans , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT
*descaf , MKL_INT *ipiv , char *equed , float *r , float *c , float *b , MKL_INT *ib ,
```

```

MKL_INT *jb , MKL_INT *descb , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
float *rcond , float *ferr , float *berr , float *work , MKL_INT *lwork , MKL_INT
*iwork , MKL_INT *liwork , MKL_INT *info );

void pdgesvx (char *fact , char *trans , MKL_INT *n , MKL_INT *nrhs , double *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *iaf , MKL_INT *jaf ,
MKL_INT *descaf , MKL_INT *ipiv , char *equed , double *r , double *c , double *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , double *rcond , double *ferr , double *berr , double *work , MKL_INT
*lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pcgesvx (char *fact , char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *iaf , MKL_INT
*jaf , MKL_INT *descaf , MKL_INT *ipiv , char *equed , float *r , float *c ,
MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *rcond , float *ferr , float *berr ,
MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pzgesvx (char *fact , char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf ,
MKL_INT *jaf , MKL_INT *descaf , MKL_INT *ipiv , char *equed , double *r , double *c ,
MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *rcond , double *ferr , double
*berr , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT *lrwork ,
MKL_INT *info );

```

Include Files

- mkl_scalapack.h

Description

The `p?gesvx` function uses the *LU* factorization to compute the solution to a real or complex system of linear equations $AX = B$, where A denotes the n -by- n submatrix $A(ia:ia+n-1, ja:ja+n-1)$, B denotes the n -by- $nrhs$ submatrix $B(ib:ib+n-1, jb:jb+nrhs-1)$ and X denotes the n -by- $nrhs$ submatrix $X(ix:ix+n-1, jx:jx+nrhs-1)$.

Error bounds on the solution and a condition estimate are also provided.

In the following description, *af* stands for the subarray of *af* from row *iaf* and column *jaf* to row *iaf*+ $n-1$ and column *jaf*+ $n-1$.

The function `p?gesvx` performs the following steps:

1. If *fact* = 'E', real scaling factors R and C are computed to equilibrate the system:

```

trans = 'N': diag(R) * A * diag(C) * diag(C)⁻¹ * X = diag(R) * B
trans = 'T': (diag(R) * A * diag(C))ᵀ * diag(R)⁻¹ * X = diag(C) * B
trans = 'C': (diag(R) * A * diag(C))ᴴ * diag(R)⁻¹ * X = diag(C) * B

```

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(R) * A * \text{diag}(C)$ and B by $\text{diag}(R) * B$ (if *trans*='N') or $\text{diag}(C) * B$ (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as $A = PLU$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. The factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than relative machine precision, steps 4 - 6 are skipped.

4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(C)$ (if $\text{trans} = 'N'$) or $\text{diag}(R)$ (if $\text{trans} = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	<p>(global) Must be 'F', 'N', or 'E'.</p> <p>Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.</p> <p>If <i>fact</i> = 'F' then, on entry, <i>af</i> and <i>ipiv</i> contain the factored form of A. If <i>equed</i> is not 'N', the matrix A has been equilibrated with scaling factors given by r and c. Arrays a, <i>af</i>, and <i>ipiv</i> are not modified.</p> <p>If <i>fact</i> = 'N', the matrix A is copied to <i>af</i> and factored.</p> <p>If <i>fact</i> = 'E', the matrix A is equilibrated if necessary, then copied to <i>af</i> and factored.</p>
<i>trans</i>	<p>(global) Must be 'N', 'T', or 'C'.</p> <p>Specifies the form of the system of equations:</p> <p>If <i>trans</i> = 'N', the system has the form $A*X = B$ (No transpose);</p> <p>If <i>trans</i> = 'T', the system has the form $A^T*X = B$ (Transpose);</p> <p>If <i>trans</i> = 'C', the system has the form $A^H*X = B$ (Conjugate transpose);</p>
<i>n</i>	<p>(global) The number of linear equations; the order of the submatrix A ($n \geq 0$).</p>
<i>nrhs</i>	<p>(global) The number of right hand sides; the number of columns of the distributed submatrices B and X ($nrhs \geq 0$).</p>
<i>a</i> , <i>af</i> , <i>b</i> , <i>work</i>	<p>(local)</p> <p>Pointers into the local memory to arrays of local size <i>a</i>: $\text{lld_a*LOCc}(ja+n-1)$, <i>af</i>: $\text{lld_af*LOCc}(ja+n-1)$, <i>b</i>: $\text{lld_b*LOCc}(jb+nrhs-1)$, <i>work</i>: <i>lwork</i>.</p> <p>The array <i>a</i> contains the matrix A. If <i>fact</i> = 'F' and <i>equed</i> is not 'N', then A must have been equilibrated by the scaling factors in r and/or c.</p> <p>The array <i>af</i> is an input argument if <i>fact</i> = 'F'. In this case it contains on entry the factored form of the matrix A, that is, the factors L and U from the factorization $A = P*L*U$ as computed by p?getrf. If <i>equed</i> is not 'N', then <i>af</i> is the factored form of the equilibrated matrix A.</p> <p>The array <i>b</i> contains on entry the matrix B whose columns are the right-hand sides for the systems of equations.</p> <p><i>work</i> is a workspace array. The size of <i>work</i> is (<i>lwork</i>).</p>
<i>ia</i> , <i>ja</i>	<p>(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A(ia:ia+n-1, ja:ja+n-1)$, respectively.</p>

<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>iaf, jaf</i>	(global) The row and column indices in the global matrix <i>AF</i> indicating the first row and the first column of the subarray <i>af</i> , respectively.
<i>descaf</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>AF</i> .
<i>ib, jb</i>	(global) The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the submatrix <i>B(ib:ib+n-1, jb:jb+nrhs-1)</i> , respectively.
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>ipiv</i>	<p>(local) Array of size $LOCr(m_a) + mb_a$.</p> <p>The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'.</p> <p>On entry, it contains the pivot indices from the factorization $A = P * L * U$ as computed by <code>p?getrf</code>; (local) row <i>i</i> of the matrix was interchanged with the (global) row <i>ipiv[i - 1]</i>.</p> <p>This array must be aligned with $A(ia:ia+n-1, *)$.</p>
<i>equed</i>	<p>(global) Must be 'N', 'R', 'C', or 'B'. <i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done:</p> <p>If <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N');</p> <p>If <i>equed</i> = 'R', row equilibration was done, that is, <i>A</i> has been premultiplied by <code>diag(r)</code>;</p> <p>If <i>equed</i> = 'C', column equilibration was done, that is, <i>A</i> has been postmultiplied by <code>diag(c)</code>;</p> <p>If <i>equed</i> = 'B', both row and column equilibration was done; <i>A</i> has been replaced by <code>diag(r) * A * diag(c)</code>.</p>
<i>r, c</i>	<p>(local)</p> <p>Arrays of size $LOCr(m_a)$ and $LOCc(n_a)$, respectively.</p> <p>The array <i>r</i> contains the row scale factors for <i>A</i>, and the array <i>c</i> contains the column scale factors for <i>A</i>. These arrays are input arguments if <i>fact</i> = 'F' only; otherwise they are output arguments. If <i>equed</i> = 'R' or 'B', <i>A</i> is multiplied on the left by <code>diag(r)</code>; if <i>equed</i> = 'N' or 'C', <i>r</i> is not accessed.</p> <p>If <i>fact</i> = 'F' and <i>equed</i> = 'R' or 'B', each element of <i>r</i> must be positive.</p> <p>If <i>equed</i> = 'C' or 'B', <i>A</i> is multiplied on the right by <code>diag(c)</code>; if <i>equed</i> = 'N' or 'R', <i>c</i> is not accessed.</p> <p>If <i>fact</i> = 'F' and <i>equed</i> = 'C' or 'B', each element of <i>c</i> must be positive. Array <i>r</i> is replicated in every process column, and is aligned with the distributed matrix <i>A</i>. Array <i>c</i> is replicated in every process row, and is aligned with the distributed matrix <i>A</i>.</p>

<i>ix, jx</i>	(global) The row and column indices in the global matrix <i>X</i> indicating the first row and the first column of the submatrix $X(ix:ix+n-1, jx:jx+nrhs-1)$, respectively.
<i>descx</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>X</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> ; must be at least $\max(p?gecon(lwork), p?gerfs(lwork)) + LOCr(n_a)$.
<i>iwork</i>	(local, psgesvx/pdgesvx only). Workspace array. The size of <i>iwork</i> is (<i>liwork</i>).
<i>liwork</i>	(local, psgesvx/pdgesvx only). The size of the array <i>iwork</i> , must be at least $LOCr(n_a)$.
<i>rwork</i>	(local) Workspace array, used in complex flavors only. The size of <i>rwork</i> is (<i>lrwork</i>).
<i>lrwork</i>	(local or global, pcgesvx/pzgesvx only). The size of the array <i>rwork</i> ; must be at least $2*LOCc(n_a)$.

Output Parameters

<i>x</i>	(local) Pointer into the local memory to an array of local size $l1d_x*LOCc(jx+nrhs-1)$. If <i>info</i> = 0, the array <i>x</i> contains the solution matrix <i>X</i> to the <i>original</i> system of equations. Note that <i>A</i> and <i>B</i> are modified on exit if <i>equed</i> ≠'N', and the solution to the <i>equilibrated</i> system is: diag(<i>C</i>)-1* <i>X</i> , if <i>trans</i> = 'N' and <i>equed</i> = 'C' or 'B'; and diag(<i>R</i>)-1* <i>X</i> , if <i>trans</i> = 'T' or 'C' and <i>equed</i> = 'R' or 'B'.
<i>a</i>	Array <i>a</i> is not modified on exit if <i>fact</i> = 'F' or 'N', or if <i>fact</i> = 'E' and <i>equed</i> = 'N'. If <i>equed</i> ≠'N', <i>A</i> is scaled on exit as follows: <i>equed</i> = 'R': $A = \text{diag}(R) * A$ <i>equed</i> = 'C': $A = A * \text{diag}(c)$ <i>equed</i> = 'B': $A = \text{diag}(R) * A * \text{diag}(c)$
<i>af</i>	If <i>fact</i> = 'N' or 'E', then <i>af</i> is an output argument and on exit returns the factors <i>L</i> and <i>U</i> from the factorization $A = P * L * U$ of the original matrix <i>A</i> (if <i>fact</i> = 'N') or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>a</i> for the form of the equilibrated matrix.
<i>b</i>	Overwritten by $\text{diag}(R) * B$ if <i>trans</i> = 'N' and <i>equed</i> = 'R' or 'B'; overwritten by $\text{diag}(c) * B$ if <i>trans</i> = 'T' and <i>equed</i> = 'C' or 'B'; not changed if <i>equed</i> = 'N'.
<i>r, c</i>	These arrays are output arguments if <i>fact</i> ≠'F'.

See the description of r , c in *Input Arguments* section.

rcond

(global).

An estimate of the reciprocal condition number of the matrix A after equilibration (if done). The function sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

ferr, *berr*

(local)

Arrays of size *LOCc*(*n_b*) each. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

Arrays *ferr* and *berr* are both replicated in every process row, and are aligned with the matrices B and X .

ipiv

If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization $A = P^*L^*U$ of the original matrix A (if *fact* = 'N') or of the equilibrated matrix A (if *fact* = 'E').

equed

If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work[0]

If *info*=0, on exit *work*[0] returns the minimum value of *lwork* required for optimum performance.

iwork[0]

If *info*=0, on exit *iwork*[0] returns the minimum value of *liwork* required for optimum performance.

rwork[0]

If *info*=0, on exit *rwork*[0] returns the minimum value of *lrwork* required for optimum performance.

info

If *info*=0, the execution is successful.

info < 0: if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*. If *info* = *i*, and $i \leq n$, then $U(i,i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed. If *info* = *i*, and $i = n + 1$, then U is nonsingular, but *rcond* is less than machine precision. The factorization has been completed, but the matrix is singular to working precision and the solution and error bounds have not been computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gbsv

Computes the solution to the system of linear equations with a general banded distributed matrix and multiple right-hand sides.

Syntax

```
void psgbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , float *a ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , float *b , MKL_INT *ib , MKL_INT *descb ,
float *work , MKL_INT *lwork , MKL_INT *info );

void pdgbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , double *a ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , double *b , MKL_INT *ib , MKL_INT
*descb , double *work , MKL_INT *lwork , MKL_INT *info );

void pcgbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , MKL_Complex8
*a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzgbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , MKL_Complex16
*a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?gbsv` function computes the solution to a real or complex system of linear equations

$$\text{sub}(A) * X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an n -by- n real/complex general banded distributed matrix with bwl subdiagonals and bwu superdiagonals, and X and $\text{sub}(B) = B(ib:ib+n-1, 1:rhs)$ are n -by- $nrhs$ distributed matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor $\text{sub}(A)$ as $\text{sub}(A) = P * L * U * Q$, where P and Q are permutation matrices, and L and U are banded lower and upper triangular matrices, respectively. The matrix Q represents reordering of columns for the sake of parallelism, while P represents reordering of rows for numerical stability using classic partial pivoting.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

n	(global) The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
bwl	(global) The number of subdiagonals within the band of A ($0 \leq bwl \leq n-1$).
bwu	(global) The number of superdiagonals within the band of A ($0 \leq bwu \leq n-1$).
$nrhs$	(global) The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).
a, b	(local) Pointers into the local memory to arrays of local size a : $lld_a * LOCC(ja + n - 1)$ and b : $lld_b * LOCC(nrhs)$.

On entry, the array *a* contains the local pieces of the global array *A*.

On entry, the array *b* contains the right hand side distributed matrix sub(*B*).

ja (global) The index in the global matrix *A* indicating the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) array of size *dlen_*. The array descriptor for the distributed matrix *A*.

If *desca*[*dtype_* - 1] = 501, then *dlen_* ≥ 7;

else if *desca*[*dtype_* - 1] = 1, then *dlen_* ≥ 9.

ib (global) The row index in the global matrix *B* indicating the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb (global and local) array of size *dlen_*. The array descriptor for the distributed matrix *B*.

If *descb*[*dtype_-*1] = 502, then *dlen_* ≥ 7;

else if *descb*[*dtype_-*1] = 1, then *dlen_* ≥ 9.

work (local)

Workspace array of size *lwork*.

lwork (local or global) The size of the array *work*, must be at least $lwork \geq (NB + b_{wu}) * (b_{wl} + b_{wu}) + 6 * (b_{wl} + b_{wu}) * (b_{wl} + 2 * b_{wu}) + \max(nr_{hs} * (NB + 2 * b_{wl} + 4 * b_{wu}), 1)$.

Output Parameters

a On exit, contains details of the factorization. Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

b On exit, this array contains the local pieces of the solution distributed matrix *X*.

ipiv (local) array.

The size of *ipiv* must be at least *desca*[*NB* - 1]. This array contains pivot indices for local factorizations. You should not alter the contents between factorization and solve.

work[0] On exit, *work*[0] contains the minimum value of *lwork* required for optimum performance.

info If *info*=0, the execution is successful. *info* < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:

If $info = k \leq NPROCS$, the submatrix stored on processor $info$ and factored locally was not nonsingular, and the factorization was not completed. If $info = k > NPROCS$, the submatrix stored on processor $info - NPROCS$ representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dbsv

Solves a general band system of linear equations.

Syntax

```
void psdbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , float *a ,
MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *work ,
MKL_INT *lwork , MKL_INT *info );

void pddbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , double *a ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *work ,
MKL_INT *lwork , MKL_INT *info );

void pcdbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , MKL_Complex8
*a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzdbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , MKL_Complex16
*a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?dbsv` function solves the following system of linear equations:

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real/complex banded diagonally dominant-like distributed matrix with bandwidth bwl , bwu .

Gaussian elimination without pivoting is used to factor a reordering of the matrix into LU .

Product and Performance Information

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

Notice revision #20201201

Input Parameters

n	(global) The order of the distributed submatrix A , ($n \geq 0$).
bwl	(global) Number of subdiagonals. $0 \leq bwl \leq n-1$.
bwu	(global) Number of subdiagonals. $0 \leq bwu \leq n-1$.

<i>nrhs</i>	(global) The number of right-hand sides; the number of columns of the distributed submatrix <i>B</i> , (<i>nrhs</i> ≥ 0).
<i>a</i>	(local). Pointer into the local memory to an array with leading size $lld_a \geq (bwl + bwu + 1)$ (stored in <i>desca</i>). On entry, this array contains the local pieces of the distributed matrix.
<i>ja</i>	(global) The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) array of size <i>dlen</i> . If 1d type (<i>dtype_a</i> =501 or 502), <i>dlen</i> ≥ 7; If 2d type (<i>dtype_a</i> =1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>A</i> . Contains information of mapping of <i>A</i> to memory.
<i>b</i>	(local) Pointer into the local memory to an array of local lead size $lld_b \geq nb$. On entry, this array contains the local pieces of the right hand sides <i>B</i> (<i>ib</i> : <i>ib</i> + <i>n</i> - 1, 1: <i>nrhs</i>).
<i>ib</i>	(global) The row index in the global matrix <i>B</i> indicating the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) array of size <i>dlen</i> . If 1d type (<i>dtype_b</i> = 502), <i>dlen</i> ≥ 7; If 2d type (<i>dtype_b</i> = 1), <i>dlen</i> ≥ 9. The array descriptor for the distributed matrix <i>B</i> . Contains information of mapping of <i>B</i> to memory.
<i>work</i>	(local). Temporary workspace. This space may be overwritten in between calls to functions. <i>work</i> must be the size given in <i>lwork</i> .
<i>lwork</i>	(local or global) Size of user-input workspace <i>work</i> . If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i> [0] and an error code is returned. $lwork \geq nb(bwl + bwu) + 6 \max(bwl, bwu) * \max(bwl, bwu) + \max(\max(bwl, bwu) nrhs, \max(bwl, bwu) * \max(bwl, bwu))$

Output Parameters

<i>a</i>	On exit, this array contains information containing details of the factorization. Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
<i>b</i>	On exit, this contains the local piece of the solutions distributed matrix <i>X</i> .

work On exit, *work*[0] contains the minimal *lwork*.

info (local) If *info*=0, the execution is successful.

< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

> 0: If *info* = *k* < NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.

If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dtsv

Solves a general tridiagonal system of linear equations.

Syntax

```
void psdtsv (MKL_INT *n , MKL_INT *nrhs , float *dl , float *d , float *du , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *work , MKL_INT
*lwork , MKL_INT *info );

void pddtsv (MKL_INT *n , MKL_INT *nrhs , double *dl , double *d , double *du , MKL_INT
*ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *work , MKL_INT
*lwork , MKL_INT *info );

void pcdtsv (MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *dl , MKL_Complex8 *d ,
MKL_Complex8 *du , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzdtsv (MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *dl , MKL_Complex16 *d ,
MKL_Complex16 *du , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The function solves a system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n complex tridiagonal diagonally dominant-like distributed matrix.

Gaussian elimination without pivoting is used to factor a reordering of the matrix into $L U$.

Product and Performance Information

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

Product and Performance Information

Notice revision #20201201

Input Parameters

<i>n</i>	(global) The order of the distributed submatrix A ($n \geq 0$).
<i>nrhs</i>	The number of right hand sides; the number of columns of the distributed matrix B ($nrhs \geq 0$).
<i>dl</i>	(local). Pointer to local part of global vector storing the lower diagonal of the matrix. Globally, $dl[0]$ is not referenced, and dl must be aligned with d . Must be of size $> desca[nb_ - 1]$.
<i>d</i>	(local). Pointer to local part of global vector storing the main diagonal of the matrix.
<i>du</i>	(local). Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, $du[n - 1]$ is not referenced, and du must be aligned with d .
<i>ja</i>	(global) The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) array of size $dlen$. If 1d type ($dtype_a=501$ or 502), $dlen \geq 7$; If 2d type ($dtype_a=1$), $dlen \geq 9$. The array descriptor for the distributed matrix A . Contains information of mapping of A to memory.
<i>b</i>	(local) Pointer into the local memory to an array of local lead size $lld_b > nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib + n - 1, 1:nrhs)$.
<i>ib</i>	(global) The row index in the global matrix B indicating the first row of the matrix to be operated on (which may be either all of b or a submatrix of B).
<i>descb</i>	(global and local) array of size $dlen$. If 1d type ($dtype_b=502$), $dlen \geq 7$; If 2d type ($dtype_b=1$), $dlen \geq 9$. The array descriptor for the distributed matrix B . Contains information of mapping of B to memory.
<i>work</i>	(local).

lwork (local or global) Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*[0] and an error code is returned. $lwork > (12 * NPCOL + 3 * nb) + \max((10 + 2 * \min(100, nrhs)) * NPCOL + 4 * nrhs, 8 * NPCOL)$

Output Parameters

d1 On exit, this array contains information containing the * factors of the matrix.

d On exit, this array contains information containing the * factors of the matrix. Must be of size $> desca[nb_ - 1]$.

du On exit, this array contains information containing the * factors of the matrix. Must be of size $> desca[nb_ - 1]$.

b On exit, this contains the local piece of the solutions distributed matrix X.

work On exit, *work*[0] contains the minimal *lwork*.

info (local) If *info*=0, the execution is successful.

< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i * 100 + j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

> 0: If *info* = $k < NPROCS$, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.

If *info* = $k > NPROCS$, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?posv

Solves a symmetric positive definite system of linear equations.

Syntax

```
void psposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT
*info );

void pdposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT
*info );

void pcposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );

void pzposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?posv` function computes the solution to a real/complex system of linear equations

$$\text{sub}(A) * X = \text{sub}(B),$$

where $\text{sub}(A)$ denotes $A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ and is an n -by- n symmetric/Hermitian distributed positive definite matrix and X and $\text{sub}(B)$ denoting $B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+nrhs-1)$ are n -by- $nrhs$ distributed matrices. The Cholesky decomposition is used to factor $\text{sub}(A)$ as

$$\text{sub}(A) = U^T * U, \text{ if } \text{uplo} = 'U', \text{ or}$$

$$\text{sub}(A) = L * L^T, \text{ if } \text{uplo} = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of $\text{sub}(A)$ is then used to solve the system of equations.

Input Parameters

<code>uplo</code>	(global) Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of $\text{sub}(A)$ is stored.
<code>n</code>	(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<code>nrhs</code>	The number of right-hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).
<code>a</code>	(local) Pointer into the local memory to an array of size $lld_a * LOCC(\text{ja}+n-1)$. On entry, this array contains the local pieces of the n -by- n symmetric distributed matrix $\text{sub}(A)$ to be factored. If <code>uplo = 'U'</code> , the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <code>uplo = 'L'</code> , the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.
<code>ia, ja</code>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<code>desca</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .
<code>b</code>	(local) Pointer into the local memory to an array of size $lld_b * LOCC(\text{jb}+nrhs-1)$. On entry, the local pieces of the right hand sides distributed matrix $\text{sub}(B)$.
<code>ib, jb</code>	(global) The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B , respectively.
<code>descb</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix B .

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, this array contains the local pieces of the factor <i>U</i> or <i>L</i> from the Cholesky factorization $\text{sub}(A) = U^H U$, or $L^* L^H$.
<i>b</i>	On exit, if <i>info</i> = 0, <i>sub(B)</i> is overwritten by the solution distributed matrix <i>X</i> .
<i>info</i>	(global) If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> -1, had an illegal value, then <i>info</i> = $-(i*100+j)$, if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$. If <i>info</i> > 0: If <i>info</i> = <i>k</i> , the leading minor of order <i>k</i> , <i>A</i> (<i>ia:ia+k-1</i> , <i>ja:ja+k-1</i>) is not positive definite, and the factorization could not be completed, and the solution has not been computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?posvx

Solves a symmetric or Hermitian positive definite system of linear equations.

Syntax

```
void psposvx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT
*descaf , char *equed , float *sr , float *sc , float *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *rcond ,
float *ferr , float *berr , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT
*liwork , MKL_INT *info );

void pdposvx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT
*descaf , char *equed , double *sr , double *sc , double *b , MKL_INT *ib , MKL_INT
*jb , MKL_INT *descb , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double
*rcond , double *ferr , double *berr , double *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *liwork , MKL_INT *info );

void pcposvx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *iaf , MKL_INT
*jaf , MKL_INT *descaf , char *equed , float *sr , float *sc , MKL_Complex8 *b , MKL_INT
*ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , float *rcond , float *ferr , float *berr , MKL_Complex8 *work ,
MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pzposvx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf , MKL_INT
*jaf , MKL_INT *descaf , char *equed , double *sr , double *sc , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , double *rcond , double *ferr , double *berr , MKL_Complex16
*work , MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );
```


Include Files

- `mkl_scalapack.h`

Description

The `p?posvx` function uses the Cholesky factorization $A=U^T*U$ or $A=L*L^T$ to compute the solution to a real or complex system of linear equations

$A(ia:ia+n-1, ja:ja+n-1)*X = B(ib:ib+n-1, jb:jb+nrhs-1)$,

where $A(ia:ia+n-1, ja:ja+n-1)$ is a n -by- n matrix and X and $B(ib:ib+n-1, jb:jb+nrhs-1)$ are n -by- $nrhs$ matrices.

Error bounds on the solution and a condition estimate are also provided.

In the following comments y denotes $Y(iy:iy+m-1, jy:jy+k-1)$, an m -by- k matrix where y can be a , af , b and x .

The function `p?posvx` performs the following steps:

1. If `fact = 'E'`, real scaling factors s are computed to equilibrate the system:

$\text{diag}(sr)*A*\text{diag}(sc)*\text{inv}(\text{diag}(sc))*X = \text{diag}(sr)*B$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(sr)*A*\text{diag}(sc)$ and B by $\text{diag}(sr)*B$.

2. If `fact = 'N'` or `'E'`, the Cholesky decomposition is used to factor the matrix A (after equilibration if `fact = 'E'`) as

$A = U^T*U$, if `uplo = 'U'`, or

$A = L*L^T$, if `uplo = 'L'`,

where U is an upper triangular matrix and L is a lower triangular matrix.

3. The factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, steps 4-6 are skipped
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(sr)$ so that it solves the original system before equilibration.

Input Parameters

<code>fact</code>	(global) Must be 'F', 'N', or 'E'. Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored. If <code>fact = 'F'</code> : on entry, <code>af</code> contains the factored form of A . If <code>equed = 'Y'</code> , the matrix A has been equilibrated with scaling factors given by s . a and <code>af</code> will not be modified. If <code>fact = 'N'</code> , the matrix A will be copied to <code>af</code> and factored. If <code>fact = 'E'</code> , the matrix A will be equilibrated if necessary, then copied to <code>af</code> and factored.
<code>uplo</code>	(global) Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored.

<i>n</i>	(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) The number of right-hand sides; the number of columns of the distributed submatrices B and X . ($nrhs \geq 0$).
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of local size $lld_a * LOCC(ja + n - 1)$. On entry, the symmetric/Hermitian matrix A, except if $fact = 'F'$ and $equed = 'Y'$, then A must contain the equilibrated matrix $\text{diag}(sr) * A * \text{diag}(sc)$.</p> <p>If $uplo = 'U'$, the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced.</p> <p>If $uplo = 'L'$, the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced. A is not modified if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$ on exit.</p>
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>af</i>	<p>(local)</p> <p>Pointer into the local memory to an array of local size $lld_af * LOCC(ja + n - 1)$.</p> <p>If $fact = 'F'$, then af is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$, in the same storage format as A. If $equed \neq 'N'$, then af is the factored form of the equilibrated matrix $\text{diag}(sr) * A * \text{diag}(sc)$.</p>
<i>iaf, jaf</i>	(global) The row and column indices in the global matrix AF indicating the first row and the first column of the submatrix AF , respectively.
<i>descaf</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix AF .
<i>equed</i>	<p>(global) Must be $'N'$ or $'Y'$.</p> <p>$equed$ is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:</p> <p>If $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$);</p> <p>If $equed = 'Y'$, equilibration was done and A has been replaced by $\text{diag}(sr) * A * \text{diag}(sc)$.</p>
<i>sr</i>	<p>(local)</p> <p>Array of size lld_a.</p> <p>The array s contains the scale factors for A. This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.</p> <p>If $equed = 'N'$, s is not accessed.</p>

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

<i>b</i>	(local) Pointer into the local memory to an array of local size $lld_b * LOCr(jb + nrhs - 1)$. On entry, the <i>n</i> -by- <i>nrhs</i> right-hand side matrix <i>B</i> .
<i>ib, jb</i>	(global) The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) Array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> .
<i>x</i>	(local) Pointer into the local memory to an array of local size $lld_x * LOCr(jx + nrhs - 1)$.
<i>ix, jx</i>	(global) The row and column indices in the global matrix <i>X</i> indicating the first row and the first column of the submatrix <i>X</i> , respectively.
<i>descx</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>X</i> .
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork = \max(p?pocon(lwork), p?porfs(lwork)) + LOCr(n_a)$. $lwork = 3 * desca[lld_ - 1]$. If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>pxerbla</i> .
<i>iwork</i>	(local) Workspace array of size <i>liwork</i> .
<i>liwork</i>	(local or global) The size of the array <i>iwork</i> . <i>liwork</i> is local input and must be at least $liwork = desca[lld_ - 1]$ $liwork = LOCr(n_a)$. If <i>liwork</i> = -1, then <i>liwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>pxerbla</i> .

Output Parameters

<i>a</i>	On exit, if <i>fact</i> = 'E' and <i>equed</i> = 'Y', <i>a</i> is overwritten by $diag(sr) * a * diag(sc)$.
<i>af</i>	If <i>fact</i> = 'N', then <i>af</i> is an output argument and on exit returns the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ of the original matrix <i>A</i> .

If *fact* = 'E', then *af* is an output argument and on exit returns the triangular factor *U* or *L* from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ of the equilibrated matrix *A* (see the description of *A* for the form of the equilibrated matrix).

equed

If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

sr

This array is an output argument if *fact* ≠ 'F'.

See the description of *sr* in *Input Arguments* section.

sc

This array is an output argument if *fact* ≠ 'F'.

See the description of *sc* in *Input Arguments* section.

b

On exit, if *equed* = 'N', *b* is not modified; if *trans* = 'N' and *equed* = 'R' or 'B', *b* is overwritten by $\text{diag}(r) * b$; if *trans* = 'T' or 'C' and *equed* = 'C' or 'B', *b* is overwritten by $\text{diag}(c) * b$.

x

(local)

If *info* = 0 the *n*-by-*nrhs* solution matrix *X* to the original system of equations.

Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the equilibrated system is

$\text{inv}(\text{diag}(sc)) * X$ if *trans* = 'N' and *equed* = 'C' or 'B', or

$\text{inv}(\text{diag}(sr)) * X$ if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'.

rcond

(global)

An estimate of the reciprocal condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision (in particular, if *rcond*=0), the matrix is singular to working precision. This condition is indicated by a return code of *info* > 0.

ferr

Arrays of size at least $\max(LOC, n_b)$. The estimated forward error bounds for each solution vector *X*(*j*) (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution, *ferr*[*j* - 1] bounds the magnitude of the largest entry in $(X(j) - xtrue)$ divided by the magnitude of the largest entry in *X*(*j*). The quality of the error bound depends on the quality of the estimate of $\text{norm}(\text{inv}(A))$ computed in the code; if the estimate of $\text{norm}(\text{inv}(A))$ is accurate, the error bound is guaranteed.

berr

(local)

Arrays of size at least $\max(LOC, n_b)$. The componentwise relative backward error of each solution vector *X*(*j*) (the smallest relative change in any entry of *A* or *B* that makes *X*(*j*) an exact solution).

work[0]

(local) On exit, *work*[0] returns the minimal and optimal *liwork*.

info

(global)

If *info*=0, the execution is successful.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value

> 0: if *info* = *i*, and *i* is ≤ *n*: if *info* = *i*, the leading minor of order *i* of *a* is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed.

= *n*+1: *rcond* is less than machine precision. The factorization has been completed, but the matrix is singular to working precision, and the solution and error bounds have not been computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pbsv

Solves a symmetric/Hermitian positive definite banded system of linear equations.

Syntax

```
void pspbsv (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , float *a , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *work , MKL_INT
*lwork , MKL_INT *info );

void pdpbsv (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , double *a , MKL_INT
*ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *work , MKL_INT
*lwork , MKL_INT *info );

void pcpbsv (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , MKL_Complex8 *a ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzpbsv (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , MKL_Complex16 *a ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?pbsv function solves a system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real/complex banded symmetric positive definite distributed matrix with bandwidth bw .

Cholesky factorization is used to factor a reordering of the matrix into $L * L'$.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

uplo (global) Must be 'U' or 'L'.
Indicates whether the upper or lower triangular of *A* is stored.

	<p>If <code>uplo = 'U'</code>, the upper triangular A is stored</p> <p>If <code>uplo = 'L'</code>, the lower triangular of A is stored.</p>
<code>n</code>	(global) The order of the distributed matrix A ($n \geq 0$).
<code>bw</code>	(global) The number of subdiagonals in L or U . $0 \leq bw \leq n-1$.
<code>nrhs</code>	(global) The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
<code>a</code>	<p>(local).</p> <p>Pointer into the local memory to an array with leading size <code>lld_a</code> $\geq (bw + 1)$ (stored in <code>desca</code>). On entry, this array contains the local pieces of the distributed matrix <code>sub(A)</code> to be factored.</p>
<code>ja</code>	(global) The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<code>desca</code>	(global and local) array of size <code>dlen</code> . The array descriptor for the distributed matrix A .
<code>b</code>	<p>(local)</p> <p>Pointer into the local memory to an array of local lead size <code>lld_b</code> $\geq nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib + n-1, 1:nrhs)$.</p>
<code>ib</code>	(global) The row index in the global matrix B indicating the first row of the matrix to be operated on (which may be either all of b or a submatrix of B).
<code>descb</code>	<p>(global and local) array of size <code>dlen</code>.</p> <p>If 1D type (<code>dtype_b = 502</code>), <code>dlen</code> ≥ 7;</p> <p>If 2D type (<code>dtype_b = 1</code>), <code>dlen</code> ≥ 9.</p> <p>The array descriptor for the distributed matrix B.</p> <p>Contains information of mapping of B to memory.</p>
<code>work</code>	<p>(local).</p> <p>Temporary workspace. This space may be overwritten in between calls to functions. <code>work</code> must be the size given in <code>lwork</code>.</p>
<code>lwork</code>	(local or global) Size of user-input workspace <code>work</code> . If <code>lwork</code> is too small, the minimal acceptable size will be returned in <code>work[0]</code> and an error code is returned. <code>lwork</code> $\geq (nb + 2 * bw) * bw + \max((bw * nrhs), bw * bw)$

Output Parameters

<code>a</code>	On exit, this array contains information containing details of the factorization. Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
<code>b</code>	On exit, contains the local piece of the solutions distributed matrix X .
<code>work</code>	On exit, <code>work[0]</code> contains the minimal <code>lwork</code> .
<code>info</code>	(global) If <code>info=0</code> , the execution is successful.

< 0 : If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

> 0 : If $info = k \leq NPROCS$, the submatrix stored on processor $info$ and factored locally was not positive definite, and the factorization was not completed.

If $info = k > NPROCS$, the submatrix stored on processor $info-NPROCS$ representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ptsv

Syntax

Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations.

```
void psptsv (MKL_INT *n , MKL_INT *nrhs , float *d , float *e , MKL_INT *ja , MKL_INT
*desca , float *b , MKL_INT *ib , MKL_INT *descb , float *work , MKL_INT *lwork ,
MKL_INT *info );

void pdptsv (MKL_INT *n , MKL_INT *nrhs , double *d , double *e , MKL_INT *ja , MKL_INT
*desca , double *b , MKL_INT *ib , MKL_INT *descb , double *work , MKL_INT *lwork ,
MKL_INT *info );

void pcptsv (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *d , MKL_Complex8 *e ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzptsv (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *d , MKL_Complex16 *e ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?ptsv function solves a system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real tridiagonal symmetric positive definite distributed matrix.

Cholesky factorization is used to factor a reordering of the matrix into $L * L'$.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>n</i>	(global) The order of matrix <i>A</i> ($n \geq 0$).
<i>nrhs</i>	(global) The number of right-hand sides; the number of columns of the distributed submatrix <i>B</i> ($nrhs \geq 0$).
<i>d</i>	(local) Pointer to local part of global vector storing the main diagonal of the matrix.
<i>e</i>	(local) Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, <i>du</i> (<i>n</i>) is not referenced, and <i>du</i> must be aligned with <i>d</i> .
<i>ja</i>	(global) The index in the global matrix <i>A</i> indicating the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) array of size <i>dlen</i> . If 1d type (<i>dtype_a</i> =501 or 502), $dlen \geq 7$; If 2d type (<i>dtype_a</i> =1), $dlen \geq 9$. The array descriptor for the distributed matrix <i>A</i> . Contains information of mapping of <i>A</i> to memory.
<i>b</i>	(local) Pointer into the local memory to an array of local lead size $lld_b \geq nb$. On entry, this array contains the local pieces of the right hand sides <i>B</i> (<i>ib:ib+n-1</i> , 1: <i>nrhs</i>).
<i>ib</i>	(global) The row index in the global matrix <i>B</i> indicating the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) array of size <i>dlen</i> . If 1d type (<i>dtype_b</i> = 502), $dlen \geq 7$; If 2d type (<i>dtype_b</i> = 1), $dlen \geq 9$. The array descriptor for the distributed matrix <i>B</i> . Contains information of mapping of <i>B</i> to memory.
<i>work</i>	(local). Temporary workspace. This space may be overwritten in between calls to functions. <i>work</i> must be the size given in <i>lwork</i> .
<i>lwork</i>	(local or global) Size of user-input workspace <i>work</i> . If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i> [0] and an error code is returned. $lwork > (12 * NPCOL + 3 * nb) + \max((10 + 2 * \min(100, nrhs)) * NPCOL + 4 * nrhs, 8 * NPCOL)$.

Output Parameters

<i>d</i>	On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to <i>desca</i> [<i>nb</i> - 1].
----------	--

<i>e</i>	On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to <i>desca</i> [<i>nb_</i> - 1].
<i>b</i>	On exit, this contains the local piece of the solutions distributed matrix <i>X</i> .
<i>work</i>	On exit, <i>work</i> [0] contains the minimal <i>lwork</i> .
<i>info</i>	(local) If <i>info</i> =0, the execution is successful. < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . > 0: If <i>info</i> = <i>k</i> ≤ NPROCS, the submatrix stored on processor <i>info</i> and factored locally was not positive definite, and the factorization was not completed. If <i>info</i> = <i>k</i> > NPROCS, the submatrix stored on processor <i>info</i> -NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gels

Solves overdetermined or underdetermined linear systems involving a matrix of full rank.

Syntax

```
void psgels (char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , float *work , MKL_INT *lwork , MKL_INT *info );

void pdgels (char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , double *work , MKL_INT *lwork , MKL_INT *info );

void pcgels (char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT
*jb , MKL_INT *descb , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzgels (char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT
*jb , MKL_INT *descb , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The *p?gels* function solves overdetermined or underdetermined real/ complex linear systems involving an *m*-by-*n* matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$, or its transpose/ conjugate-transpose, using a *QTQ* or *LQ* factorization of $\text{sub}(A)$. It is assumed that $\text{sub}(A)$ has full rank.

The following options are provided:

1. If *trans* = 'N' and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

minimize $||\text{sub}(B) - \text{sub}(A) * X||$

2. If $\text{trans} = 'N'$ and $m < n$: find the minimum norm solution of an underdetermined system $\text{sub}(A) * X = \text{sub}(B)$.
3. If $\text{trans} = 'T'$ and $m \geq n$: find the minimum norm solution of an undetermined system $\text{sub}(A)^T * X = \text{sub}(B)$.
4. If $\text{trans} = 'T'$ and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

minimize $||\text{sub}(B) - \text{sub}(A)^T * X||$,

where $\text{sub}(B)$ denotes $B(ib:ib+m-1, jb:jb+nrhs-1)$ when $\text{trans} = 'N'$ and $B(ib:ib+n-1, jb:jb+nrhs-1)$ otherwise. Several right hand side vectors b and solution vectors x can be handled in a single call; when $\text{trans} = 'N'$, the solution vectors are stored as the columns of the n -by- $nrhs$ right hand side matrix $\text{sub}(B)$ and the m -by- $nrhs$ right hand side matrix $\text{sub}(B)$ otherwise.

Input Parameters

<i>trans</i>	(global) Must be 'N', or 'T'. If $\text{trans} = 'N'$, the linear system involves matrix $\text{sub}(A)$; If $\text{trans} = 'T'$, the linear system involves the transposed matrix A^T (for real flavors only).
<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$ ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) The number of right-hand sides; the number of columns in the distributed submatrices $\text{sub}(B)$ and X . ($nrhs \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$. On entry, contains the m -by- n matrix A .
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>b</i>	(local) Pointer into the local memory to an array of local size $lld_b * LOCC(jb + nrhs - 1)$. On entry, this array contains the local pieces of the distributed matrix B of right-hand side vectors, stored columnwise; $\text{sub}(B)$ is m -by- $nrhs$ if $\text{trans} = 'N'$, and n -by- $nrhs$ otherwise.
<i>ib, jb</i>	(global) The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B , respectively.
<i>descb</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix B .
<i>work</i>	(local) Workspace array with size $lwork$.
<i>lwork</i>	(local or global) .

The size of the array *work/work* is local input and must be at least $lwork \geq ltau + \max(lwf, lws)$, where if $m > n$, then

```
ltau = numroc(ja+min(m,n)-1, nb_a, MYCOL, csrc_a, NPCOL),
lwf = nb_a*(mpa0 + nqa0 + nb_a)
lws = max((nb_a*(nb_a-1))/2, (nrhsqb0 + mpb0)*nb_a) +
nb_a*nb_a
else
ltau = numroc(ia+min(m,n)-1, mb_a, MYROW, rsrc_a, NPROW),
lwf = mb_a * (mpa0 + nqa0 + mb_a)
lws = max((mb_a*(mb_a-1))/2, (npb0 + max(nqa0 +
numroc(numroc(n+iroffb, mb_a, 0, 0, NPROW), mb_a, 0, 0,
lcmp), nrhsqb0))*mb_a) + mb_a*mb_a
end if,
```

where $lcmp = lcm/NPROW$ with $lcm = ilcm(NPROW, NPCOL)$,

```
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYROW, rsrc_a, NPROW)
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL),
iroffb = mod(ib-1, mb_b),
icoffb = mod(jb-1, nb_b),
ibrow = indxg2p(ib, mb_b, MYROW, rsrc_b, NPROW),
ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL),
mpb0 = numroc(m+iroffb, mb_b, MYROW, icrow, NPROW),
nqb0 = numroc(n+icoffb, nb_b, MYCOL, ibcol, NPCOL),
```

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

ilcm, *indxg2p* and *numroc* are ScaLAPACK tool functions; *MYROW*, *MYCOL*, *NPROW*, and *NPCOL* can be determined by calling the function *blacs_gridinfo*.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

<code>a</code>	On exit, If $m \geq n$, <code>sub(A)</code> is overwritten by the details of its QR factorization as returned by <code>p?geqrf</code> ; if $m < n$, <code>sub(A)</code> is overwritten by details of its LQ factorization as returned by <code>p?gelqf</code> .
<code>b</code>	<p>On exit, <code>sub(B)</code> is overwritten by the solution vectors, stored columnwise: if <code>trans = 'N'</code> and $m \geq n$, rows 1 to n of <code>sub(B)</code> contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements $n+1$ to m in that column;</p> <p>If <code>trans = 'N'</code> and $m < n$, rows 1 to n of <code>sub(B)</code> contain the minimum norm solution vectors;</p> <p>If <code>trans = 'T'</code> and $m \geq n$, rows 1 to m of <code>sub(B)</code> contain the minimum norm solution vectors; if <code>trans = 'T'</code> and $m < n$, rows 1 to m of <code>sub(B)</code> contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements $m+1$ to n in that column.</p>
<code>work[0]</code>	On exit, <code>work[0]</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	<p>(global)</p> <p>= 0: the execution is successful.</p> <p>< 0: if the i-th argument is an array and the j-entry had an illegal value, then <code>info</code> = - ($i * 100 + j$), if the i-th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?syev

Computes selected eigenvalues and eigenvectors of a symmetric matrix.

Syntax

```
void pssyev (char *jobz , char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *w , float *z , MKL_INT *iz , MKL_INT *jz , MKL_INT
*descz , float *work , MKL_INT *lwork , MKL_INT *info );

void pdsyev (char *jobz , char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *w , double *z , MKL_INT *iz , MKL_INT *jz , MKL_INT
*descz , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?syev` function computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A by calling the recommended sequence of ScaLAPACK functions.

In its present form, the function assumes a homogeneous system and makes no checks for consistency of the eigenvalues or eigenvectors across the different processes. Because of this, it is possible that a heterogeneous system may return incorrect results without any error messages.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

<i>jobz</i>	(global) Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	(global) Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored: If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	(global) The number of rows and columns of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	(local) Block cyclic array of global size $n*n$ and local size $lld_a*LOCc(ja+n-1)$. On entry, the symmetric matrix <i>A</i> . If <i>uplo</i> = 'U', only the upper triangular part of <i>A</i> is used to define the elements of the symmetric matrix. If <i>uplo</i> = 'L', only the lower triangular part of <i>A</i> is used to define the elements of the symmetric matrix.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>iz, jz</i>	(global) The row and column indices in the global matrix <i>Z</i> indicating the first row and the first column of the submatrix <i>Z</i> , respectively.
<i>descz</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	(local) Array of size <i>lwork</i> .
<i>lwork</i>	(local) See below for definitions of variables used to define <i>lwork</i> . If no eigenvectors are requested (<i>jobz</i> = 'N'), then $lwork \geq 5*n + sizesytrd + 1$, where <i>sizesytrd</i> is the workspace for p?sytrd and is $\max(NB*(np + 1), 3*NB)$. If eigenvectors are requested (<i>jobz</i> = 'V') then the amount of workspace required to guarantee that all eigenvectors are computed is: $qrmem = 2*n-2$

```
lworkmin = 5*n + n*ldc + max(sizemqrleft, qrmem) + 1
```

Variable definitions:

```
nb = desca[mb_ - 1] = desca[nb_ - 1] = descz[mb_ - 1] =
descz[nb_ - 1];
```

```
nn = max(n, nb, 2);
```

```
desca[rsrc_ - 1] = desca[rsrc_ - 1] = descz[rsrc_ - 1] =
descz[csrc_ - 1] = 0
```

```
np = numroc(nn, nb, 0, 0, NPROW)
```

```
nq = numroc(max(n, nb, 2), nb, 0, 0, NPCOL)
```

```
nrc = numroc(n, nb, myprowc, 0, NPROCS)
```

```
ldc = max(1, nrc)
```

sizemqrleft is the workspace for `p?ormtr` when its *side* argument is 'L'.

myprowc is defined when a new context is created as follows:

```
call blacs_get(desca[ctxt_ - 1], 0, contextc)
```

```
call blacs_gridinit(contextc, 'R', NPROCS, 1)
```

```
call blacs_gridinfo(contextc, nprowc, npcolc, myprowc,
mypcolc)
```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

On exit, the lower triangle (if *uplo*='L') or the upper triangle (if *uplo*='U') of *A*, including the diagonal, is destroyed.

w

(global).

Array of size *n*.

On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z

(local).

Array, global size $n \times n$, local size $lld_z \times LOCC(jz+n-1)$. If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues.

If *jobz* = 'N', then *z* is not referenced.

work[0]

On output, *work*[0] returns the workspace needed to guarantee completion. If the input parameters are incorrect, *work*[0] may also be incorrect.

If *jobz* = 'N' *work*[0] = minimal (optimal) amount of workspace

If *jobz* = 'V' *work*[0] = minimal workspace required to generate all the eigenvectors.

info (global)

If *info* = 0, the execution is successful.

If *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i*100+j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

If *info* > 0:

If *info* = 1 through *n*, the *i*-th eigenvalue did not converge in *steqr2* after a total of 30*n* iterations.

If *info* = *n*+1, then *p?syev* has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from *p?syev* cannot be guaranteed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?syevd

Computes all eigenvalues and eigenvectors of a real symmetric matrix by using a divide and conquer algorithm.

Syntax

```
void pssyevd (char *jobz , char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *w , float *z , MKL_INT *iz , MKL_INT *jz , MKL_INT
*descz , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT
*info );

void pdsyevd (char *jobz , char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *w , double *z , MKL_INT *iz , MKL_INT *jz , MKL_INT
*descz , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT
*info );
```

Include Files

- `mkl_scalapack.h`

Description

The *p?syevd* function computes all eigenvalues and eigenvectors of a real symmetric matrix *A* by using a divide and conquer algorithm.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global) Must be 'N' or 'V'.

Specifies if it is necessary to compute the eigenvectors:

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo (global) Must be 'U' or 'L'.

	Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: If $uplo = 'U'$, a stores the upper triangular part of A . If $uplo = 'L'$, a stores the lower triangular part of A .
n	(global) The number of rows and columns of the matrix A ($n \geq 0$).
a	(local). Block cyclic array of global size $n*n$ and local size $lld_a*LOCc(ja+n-1)$. On entry, the symmetric matrix A . If $uplo = 'U'$, only the upper triangular part of A is used to define the elements of the symmetric matrix. If $uplo = 'L'$, only the lower triangular part of A is used to define the elements of the symmetric matrix.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A . If $desca[ctxt_ - 1]$ is incorrect, <code>p?syevd</code> cannot guarantee correct error reporting.
iz, jz	(global) The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z , respectively.
$descz$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix Z . $descz[ctxt_ - 1]$ must equal $desca[ctxt_ - 1]$.
$work$	(local). Array of size $lwork$.
$lwork$	(local) The size of the array $work$. If eigenvalues are requested: $lwork \geq \max(1 + 6*n + 2*np*nq, trilwmin) + 2*n$ with $trilwmin = 3*n + \max(nb*(np + 1), 3*nb)$ $np = \text{numroc}(n, nb, myrow, iarow, NPROW)$ $nq = \text{numroc}(n, nb, mycol, iacol, NPCOL)$ If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by <code>p?xerbla</code> .
$iwork$	(local) Workspace array of size $liwork$.
$liwork$	(local) , size of $iwork$. $liwork = 7*n + 8*npcol + 2.$

Output Parameters

<code>a</code>	On exit, the lower triangle (if <code>uplo = 'L'</code>), or the upper triangle (if <code>uplo = 'U'</code>) of A , including the diagonal, is overwritten.
<code>w</code>	(global). Array of size n . If <code>info = 0</code> , w contains the eigenvalues in the ascending order.
<code>z</code>	(local). Array, global size (n, n) , local size <code>lld_z*LOCc(jz+n-1)</code> . The z parameter contains the orthonormal eigenvectors of the matrix A .
<code>work[0]</code>	On exit, returns adequate workspace to allow optimal performance.
<code>iwork[0]</code>	(local). On exit, if <code>liwork > 0</code> , <code>iwork[0]</code> returns the optimal <code>liwork</code> .
<code>info</code>	(global) If <code>info = 0</code> , the execution is successful. If <code>info < 0</code> : If the i -th argument is an array and the j -entry had an illegal value, then <code>info = -(i*100+j)</code> . If the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> . If <code>info > 0</code> : The algorithm failed to compute the <code>info/(n+1)</code> -th eigenvalue while working on the submatrix lying in global rows and columns <code>mod(info, n+1)</code> .

NOTE

`mod(x, y)` is the integer remainder of x/y .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?syevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using Relatively Robust Representation.

Syntax

```
void pssyevr(char* jobz, char* range, char* uplo, MKL_INT* n, float* a, MKL_INT* ia,
MKL_INT* ja, MKL_INT* desca, float* vl, float* vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m,
MKL_INT* nz, float* w, float* z, MKL_INT* iz, MKL_INT* jz, MKL_INT* descz, float* work,
MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);
```

```
void pdsyevr(char* jobz, char* range, char* uplo, MKL_INT* n, double* a, MKL_INT* ia,
MKL_INT* ja, MKL_INT* desca, double* vl, double* vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m,
MKL_INT* nz, double* w, double* z, MKL_INT* iz, MKL_INT* jz, MKL_INT* descz, double* work,
MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);
```

Include Files

- `mkl_scalapack.h`

Description

`p?syevr` computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A distributed in 2D blockcyclic format by calling the recommended sequence of ScaLAPACK functions.

First, the matrix A is reduced to real symmetric tridiagonal form. Then, the eigenproblem is solved using the parallel MRRR algorithm. Last, if eigenvectors have been computed, a backtransformation is done.

Upon successful completion, each processor stores a copy of all computed eigenvalues in w . The eigenvector matrix z is stored in 2D block-cyclic format distributed over all processors.

Note that subsets of eigenvalues/vectors can be selected by specifying a range of values or a range of indices for the desired eigenvalues.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<code>jobz</code>	(global) Specifies whether or not to compute the eigenvectors: = 'N': Compute eigenvalues only. = 'V': Compute eigenvalues and eigenvectors.
<code>range</code>	(global) = 'A': all eigenvalues will be found. = 'V': all eigenvalues in the interval $[vl, vu]$ will be found. = 'I': the il -th through iu -th eigenvalues will be found.
<code>uplo</code>	(global) Specifies whether the upper or lower triangular part of the symmetric matrix A is stored: = 'U': Upper triangular = 'L': Lower triangular
<code>n</code>	(global) The number of rows and columns of the matrix a . $n \geq 0$
<code>a</code>	Block cyclic array of global size $n * n$, local size $lld_a * LOC_c(ja+n-1)$. This array contains the local pieces of the symmetric distributed matrix A . If <code>uplo = 'U'</code> , only the upper triangular part of a is used to define the elements of the symmetric matrix. If <code>uplo = 'L'</code> , only the lower triangular part of a is used to define the elements of the symmetric matrix. On exit, the lower triangle (if <code>uplo='L'</code>) or the upper triangle (if <code>uplo='U'</code>) of a , including the diagonal, is destroyed.

<i>ia</i>	(global) Global row index in the global matrix <i>A</i> that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>ja</i>	(global) Global column index in the global matrix <i>A</i> that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>desca</i>	(global and local) array of size <i>dlen_</i> =9. The array descriptor for the distributed matrix <i>a</i> .
<i>vl</i>	(global) If <i>range</i> ='V', the lower bound of the interval to be searched for eigenvalues. Not referenced if <i>range</i> = 'A' or 'I'.
<i>vu</i>	(global) If <i>range</i> ='V', the upper bound of the interval to be searched for eigenvalues. Not referenced if <i>range</i> = 'A' or 'I'.
<i>il</i>	(global) If <i>range</i> ='I', the index (from smallest to largest) of the smallest eigenvalue to be returned. $il \geq 1$. Not referenced if <i>range</i> = 'A'.
<i>iu</i>	(global) If <i>range</i> ='I', the index (from smallest to largest) of the largest eigenvalue to be returned. $\min(il, n) \leq iu \leq n$. Not referenced if <i>range</i> = 'A'.
<i>iz</i>	(global) Global row index in the global matrix <i>Z</i> that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>jz</i>	(global) Global column index in the global matrix <i>Z</i> that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>descz</i>	array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>z</i> . The context <i>descz</i> [<i>ctxt_</i> - 1] must equal <i>desca</i> [<i>ctxt_</i> - 1]. Also note the array alignment requirements specified below.
<i>work</i>	(local workspace) array of size <i>lwork</i>
<i>lwork</i>	(local) Size of <i>work</i> , must be at least 3.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N') then

$$lwork \geq 2 + 5*n + \max(12 * nn, neig * (np0 + 1))$$

If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required is:

$$lwork \geq 2 + 5*n + \max(18*nn, np0 * mq0 + 2 * neig * neig) + (2 + \text{iceil}(neig, nprow*npcol))*nn$$

Variable definitions:

neig = number of eigenvectors requested

nb = *desca*[*mb_* - 1] = *desca*(*nb_*) = *descz*[*mb_* - 1] = *descz*(*nb_*)

nn = max(*n*, *neig*, 2)

desca[*rsrc_* - 1] = *desca*[*csrc_nb_* - 1] = *descz*[*rsrc_* - 1] = *descz*[*csrc_* - 1] = 0

np0 = numroc(*nn*, *neig*, 0, 0, *nprow*)

mq0 = numroc(max(*neig*, *neig*, 2), *neig*, 0, 0, *npcol*)

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x/y*), and *nprow* and *npcol* can be determined by calling the function *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by [pxerbla](#).

liwork

(local)

size of *iwork*

Let *nnp* = max(*n*, *nprow*npcol* + 1, 4). Then:

liwork ≥ 12**nnp* + 2**n* when the eigenvectors are desired

liwork ≥ 10**nnp* + 2**n* when only the eigenvalues have to be computed

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

OUTPUT Parameters

m

(global)

Total number of eigenvalues found. $0 \leq m \leq n$.

nz

(global)

Total number of eigenvectors computed. $0 \leq nz \leq m$.

The number of columns of *z* that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

	If <i>jobz</i> = 'V', <i>nz</i> = <i>m</i>
<i>w</i>	(global) array of size <i>n</i> Upon successful exit, the first <i>m</i> entries contain the selected eigenvalues in ascending order.
<i>z</i>	Block-cyclic array, global size <i>n</i> * <i>n</i> , local size <i>lld_z</i> * <i>LOC_c</i> (<i>jz</i> + <i>n</i> -1). On exit, contains local pieces of distributed matrix <i>Z</i> .
<i>work</i>	On return, <i>work</i> [0] contains the optimal amount of workspace required for efficient execution. If <i>jobz</i> ='N' <i>work</i> [0] = optimal amount of workspace required to compute the eigenvalues. If <i>jobz</i> ='V' <i>work</i> [0] = optimal amount of workspace required to compute eigenvalues and eigenvectors.
<i>iwork</i>	(local workspace) array On return, <i>iwork</i> [0] contains the amount of integer workspace required.
<i>info</i>	(global) = 0: successful exit < 0: If the <i>i</i> -th argument is an array and the <i>j</i> th-entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Application Notes

The distributed submatrices *a*(*ia**, *ja**) and *z*(*iz*:*iz*+*m*-1, *jz*:*jz*+*n*-1) must satisfy the following alignment properties:

1. Identical (quadratic) dimension: *desca*[*m* - 1] = *descz*[*m* - 1] = *desca*[*n* - 1] = *descz*[*n* - 1]
2. Quadratic conformal blocking: *desca*[*mb* - 1] = *desca*[*nb* - 1] = *descz*[*mb* - 1] = *descz*[*nb* - 1],
desca[*rsrc* - 1] = *descz*[*rsrc* - 1]
3. mod(*ia*-1, *mb_a*) = mod(*iz*-1, *mb_z*) = 0

NOTE

mod(*x*, *y*) is the integer remainder of *x*/*y*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

```
void pssyevx (char *jobz , char *range , char *uplo , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *vl , float *vu , MKL_INT *il , MKL_INT *iu ,
float *abstol , MKL_INT *m , MKL_INT *nz , float *w , float *orfac , float *z , MKL_INT
*iz , MKL_INT *jz , MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *liwork , MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT *info );
```

```
void pdsyevx (char *jobz , char *range , char *uplo , MKL_INT *n , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *vl , double *vu , MKL_INT *il , MKL_INT
*iu , double *abstol , MKL_INT *m , MKL_INT *nz , double *w , double *orfac , double
*z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , double *work , MKL_INT *lwork ,
MKL_INT *iwork , MKL_INT *liwork , MKL_INT *ifail , MKL_INT *iclustr , double *gap ,
MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `pdsyevx` function computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A by calling the recommended sequence of ScaLAPACK functions. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

<i>jobz</i>	(global) Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	(global) Must be 'A', 'V', or 'I'. If <i>range</i> = 'A', all eigenvalues will be found. If <i>range</i> = 'V', all eigenvalues in the half-open interval $[vl, vu]$ will be found. If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	(global) Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored: If <i>uplo</i> = 'U', a stores the upper triangular part of A . If <i>uplo</i> = 'L', a stores the lower triangular part of A .
<i>n</i>	(global) The number of rows and columns of the matrix A ($n \geq 0$).
<i>a</i>	(local). Block cyclic array of global size $n*n$ and local size $lld_a*LOCc(ja+n-1)$. On entry, the symmetric matrix A . If <i>uplo</i> = 'U', only the upper triangular part of A is used to define the elements of the symmetric matrix.

If `uplo = 'L'`, only the lower triangular part of A is used to define the elements of the symmetric matrix.

`ia, ja`

(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.

`desca`

(global and local) array of size `dlen_`. The array descriptor for the distributed matrix A .

`vl, vu`

(global)

If `range = 'V'`, the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if `range = 'A'` or `'I'`.

`il, iu`

(global)

If `range = 'I'`, the indices of the smallest and largest eigenvalues to be returned.

Constraints: $il \geq 1$

$\min(il, n) \leq iu \leq n$

Not referenced if `range = 'A'` or `'V'`.

`abstol`

(global).

If `jobz = 'V'`, setting `abstol` to `p?lamch(context, 'U')` yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to

$abstol + eps * \max(|a|, |b|)$,

where `eps` is the machine precision. If `abstol` is less than or equal to zero, then `eps*norm(T)` will be used in its place, where `norm(T)` is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

Eigenvalues will be computed most accurately when `abstol` is set to twice the underflow threshold `2*p?lamch('S')` not zero. If this function returns with `(mod(info,2) \neq 0)` or `(mod(info/8,2) \neq 0)`, indicating that some eigenvalues or eigenvectors did not converge, try setting `abstol` to `2*p?lamch('S')`.

`orfac`

(global).

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within `tol=orfac*norm(A)` of each other are to be reorthogonalized. However, if the workspace is insufficient (see `lwork`), `tol` may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if `orfac` equals zero. A default value of `1.0e-3` is used if `orfac` is negative. `orfac` should be identical on all processes.

`iz, jz`

(global) The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z , respectively.

`descz`

(global and local) array of size `dlen_`. The array descriptor for the distributed matrix Z . `descz[ctxt_ - 1]` must equal `desca[ctxt_ - 1]`.

work (local)
Array of size *lwork*.

lwork (local) The size of the array *work*.
See below for definitions of variables used to define *lwork*.
If no eigenvectors are requested (*jobz* = 'N'), then $lwork \geq 5*n + \max(5*nn, NB*(np0 + 1))$.
If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:
$$lwork \geq 5*n + \max(5*nn, np0*mq0 + 2*NB*NB) + \text{iceil}(neig, NPROW*NPCOL)*nn$$

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following to *lwork*:
$$(clustersize-1)*n,$$

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:
$$\{w[k-1], \dots, w[k+clustersize-2] | w[j] \leq w[j-1]\} + orfac*2*norm(A),$$

where
neig = number of eigenvectors requested
$$nb = desca[mb_ - 1] = desca[nb_ - 1] = descz[mb_ - 1] = descz[nb_ - 1];$$

$$nn = \max(n, nb, 2);$$

$$desca[rsrc_ - 1] = desca[nb_ - 1] = descz[rsrc_ - 1] = descz[csrc_ - 1] = 0;$$

$$np0 = \text{numroc}(nn, nb, 0, 0, NPROW);$$

$$mq0 = \text{numroc}(\max(neig, nb, 2), nb, 0, 0, NPCOL)$$

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)
If *lwork* is too small to guarantee orthogonality, *p?syevx* attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.
If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.
Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, *p?sygvx* computes the eigenvalues and as many eigenvectors as possible.
Relationship between workspace, orthogonality & performance:
Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:


```
lwork ≥ max(lwork, 5*n + nsytrd_lwopt),
```

where *lwork*, as defined previously, depends upon the number of eigenvectors requested, and

```
nsytrd_lwopt = n + 2*(anb+1)*(4*nps+2) + (nps + 3)*nps;
```

```
anb = pjlacenv(desca[ctxt_ - 1], 3, 'p?sytrd', 'L', 0, 0, 0, 0);
```

```
sqnpc = int(sqrt(dble(NPROW * NPCOL)));
```

```
nps = max(numroc(n, 1, 0, 0, sqnpc), 2*anb);
```

numroc is a ScaLAPACK tool functions;

pjlacenv is a ScaLAPACK environmental inquiry function

MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

For large *n*, no extra workspace is needed, however the biggest boost in performance comes for small *n*, so it is wise to provide the extra workspace (typically less than a megabyte per process).

If *clustersize* > *n*/sqrt(NPROW*NPCOL), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, *clustersize* = *n*-1) p?stein will perform no better than ?stein on single processor.

For *clustersize* = *n*/sqrt(NPROW*NPCOL) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize*>*n*/sqrt(NPROW*NPCOL) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxebla.

iwork

(local) Workspace array.

liwork

(local) , size of *iwork*. *liwork* ≥ 6**nnp*

Where: *nnp* = max(*n*, NPROW*NPCOL + 1, 4)

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxebla.

Output Parameters

a

On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m

(global) The total number of eigenvalues found; 0 ≤ *m* ≤ *n*.

<i>nz</i>	<p>(global) Total number of eigenvectors computed. $0 \leq nz \leq m$.</p> <p>The number of columns of <i>z</i> that are filled.</p> <p>If <i>jobz</i> \neq 'V', <i>nz</i> is not referenced.</p> <p>If <i>jobz</i> = 'V', <i>nz</i> = <i>m</i> unless the user supplies insufficient space and <i>p?syevx</i> is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in <i>z</i> ($m \leq descz[n_ - 1]$) and sufficient workspace to compute them. (See <i>lwork</i>). <i>p?syevx</i> is always able to detect insufficient space without computation unless <i>range</i> = 'V'.</p>
<i>w</i>	<p>(global).</p> <p>Array of size <i>n</i>. The first <i>m</i> elements contain the selected eigenvalues in ascending order.</p>
<i>z</i>	<p>(local).</p> <p>Array, global size $n*n$, local size $lld_z*LOCc(jz+n-1)$.</p> <p>If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i>.</p> <p>If <i>jobz</i> = 'N', then <i>z</i> is not referenced.</p>
<i>work</i> [0]	<p>On exit, returns workspace adequate workspace to allow optimal performance.</p>
<i>iwork</i> [0]	<p>On return, <i>iwork</i>[0] contains the amount of integer workspace required.</p>
<i>ifail</i>	<p>(global).</p> <p>Array of size <i>n</i>.</p> <p>If <i>jobz</i> = 'V', then on normal exit, the first <i>m</i> elements of <i>ifail</i> are zero. If $(\text{mod}(\text{info}, 2) \neq 0)$ on exit, then <i>ifail</i> contains the indices of the eigenvectors that failed to converge.</p> <p>If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.</p>
<i>iclustr</i>	<p>(global) Array of size $(2*NPROW*NPCOL)$</p> <p>This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see <i>lwork</i>, <i>orfac</i> and <i>info</i>). Eigenvectors corresponding to clusters of eigenvalues indexed <i>iclustr</i>(2*i-1) to <i>iclustr</i>(2*i), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. <i>iclustr</i> is a zero terminated array. <i>iclustr</i>[2*k - 1] \neq 0 and <i>iclustr</i>[2*k] = 0 if and only if <i>k</i> is the number of clusters.</p> <p><i>iclustr</i> is not referenced if <i>jobz</i> = 'N'.</p>
<i>gap</i>	<p>(global)</p> <p>Array of size $NPROW*NPCOL$</p>

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*th cluster may be as high as $(C*n) / gap[i - 1]$ where *C* is a small constant.

info

(global)

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i*100+j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

If *info* > 0: if $(\text{mod}(\text{info}, 2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure $abstol=2.0*p?lamch('U')$.

If $(\text{mod}(\text{info}/2, 2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(\text{info}/4, 2) \neq 0)$, then space limit prevented *p?syevxf* from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If $(\text{mod}(\text{info}/8, 2) \neq 0)$, then *p?stebz* failed to compute eigenvalues. Ensure $abstol=2.0*p?lamch('U')$.

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?heev

Computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix.

Syntax

```
void pcheev (char *jobz , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *w , MKL_Complex8 *z , MKL_INT *iz , MKL_INT *jz ,
MKL_INT *descz , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *lrwork ,
MKL_INT *info );
```

```
void pzheev (char *jobz , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *w , MKL_Complex16 *z , MKL_INT *iz , MKL_INT
*jz , MKL_INT *descz , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT
*lrwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?heev` function computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A by calling the recommended sequence of ScaLAPACK functions. The function assumes a homogeneous system and makes spot checks of the consistency of the eigenvalues across the different processes. A heterogeneous system may return incorrect results without any error messages.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

$jobz$	(global) Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If $jobz = 'N'$, then only eigenvalues are computed. If $jobz = 'V'$, then eigenvalues and eigenvectors are computed.
$uplo$	(global) Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: If $uplo = 'U'$, a stores the upper triangular part of A . If $uplo = 'L'$, a stores the lower triangular part of A .
n	(global) The number of rows and columns of the matrix A ($n \geq 0$).
a	(local). Block cyclic array of global size $n*n$ and local size $lld_a*LOCc(ja+n-1)$. On entry, the Hermitian matrix A . If $uplo = 'U'$, only the upper triangular part of A is used to define the elements of the Hermitian matrix. If $uplo = 'L'$, only the lower triangular part of A is used to define the elements of the Hermitian matrix.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A . If $desca[ctxt_ - 1]$ is incorrect, <code>p?heev</code> cannot guarantee correct error reporting.
iz, jz	(global) The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z , respectively.
$descz$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix Z . $descz[ctxt_ - 1]$ must equal $desca[ctxt_ - 1]$.
$work$	(local). Array of size $lwork$.
$lwork$	(local) The size of the array $work$. If only eigenvalues are requested ($jobz = 'N'$): $lwork \geq \max(nb*(np0 + 1), 3) + 3*n$

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required:

$$lwork \geq (np0+nq0+nb)*nb + 3*n + n^2$$

with $nb = desca[mb_ - 1] = desca[nb_ - 1] = nb = descz[mb_ - 1] = descz[nb_ - 1]$

$np0 = \text{numroc}(nn, nb, 0, 0, \text{NPROW})$.

$nq0 = \text{numroc}(\max(n, nb, 2), nb, 0, 0, \text{NPCOL})$.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by *pxerbla*.

rwork

(local).

Workspace array of size *lwork*.

lwork

(local) The size of the array *rwork*.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then $lwork \geq 2*n$.

If eigenvectors are requested (*jobz* = 'V'), then $lwork \geq 2*n + 2*n-2$.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the minimum size required for the *rwork* array. The required workspace is returned as the first element of *rwork*, and no error message is issued by *pxerbla*.

Output Parameters

a

On exit, the lower triangle (if *uplo* = 'L'), or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

w

(global).

Array of size *n*. The first *m* elements contain the selected eigenvalues in ascending order.

z

(local).

Array, global size $n*n$, local size $lld_z * LOCC(jz+n-1)$.

If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

work[0]

On exit, returns adequate workspace to allow optimal performance.

If *jobz* = 'N', then *work*[0] = minimal workspace only for eigenvalues.

If *jobz* = 'V', then *work*[0] = minimal workspace required to generate all the eigenvectors.

`rwork[0]` (local)
On output, `rwork[0]` returns workspace required to guarantee completion.

`info` (global)
If `info = 0`, the execution is successful.
If `info < 0`:
If the i -th argument is an array and the j -entry had an illegal value, then `info = -(i*100+j)`. If the i -th argument is a scalar and had an illegal value, then `info = -i`.
If `info > 0`:
If `info = 1` through n , the i -th eigenvalue did not converge in `?steqr2` after a total of $30*n$ iterations.
If `info = n+1`, then `p?heevd` detected heterogeneity, and the accuracy of the results cannot be guaranteed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?heevd

Computes all eigenvalues and eigenvectors of a complex Hermitian matrix by using a divide and conquer algorithm.

Syntax

```
void pcheevd (char *jobz , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *w , MKL_Complex8 *z , MKL_INT *iz , MKL_INT *jz ,
MKL_INT *descz , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *lrwork ,
MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pzheevd (char *jobz , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *w , MKL_Complex16 *z , MKL_INT *iz , MKL_INT
*jz , MKL_INT *descz , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT
*lrwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?heevd` function computes all eigenvalues and eigenvectors of a complex Hermitian matrix A by using a divide and conquer algorithm.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

`jobz` (global) Must be 'N' or 'V'.
Specifies if it is necessary to compute the eigenvectors:
If `jobz = 'N'`, then only eigenvalues are computed.

	If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	(global) Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is stored: If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	(global) The number of rows and columns of the matrix <i>A</i> ($n \geq 0$).
<i>a</i>	(local). Block cyclic array of global size $n*n$ and local size $lld_a*LOCc(ja+n-1)$. On entry, the Hermitian matrix <i>A</i> . If <i>uplo</i> = 'U', only the upper triangular part of <i>A</i> is used to define the elements of the Hermitian matrix. If <i>uplo</i> = 'L', only the lower triangular part of <i>A</i> is used to define the elements of the Hermitian matrix.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If <i>desca</i> [<i>ctxt_</i> - 1] is incorrect, p?heevd cannot guarantee correct error reporting.
<i>iz, jz</i>	(global) The row and column indices in the global matrix <i>Z</i> indicating the first row and the first column of the submatrix <i>Z</i> , respectively.
<i>descz</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> . <i>descz</i> [<i>ctxt_</i> - 1] must equal <i>desca</i> [<i>ctxt_</i> - 1].
<i>work</i>	(local). Array of size <i>lwork</i> .
<i>lwork</i>	(local) The size of the array <i>work</i> . If eigenvalues are requested: $lwork = n + (nb0 + mq0 + nb)*nb$ with $np0 = \text{numroc}(\max(n, nb, 2), nb, 0, 0, \text{NPROW})$ $mq0 = \text{numroc}(\max(n, nb, 2), nb, 0, 0, \text{NPCOL})$ If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by p?xerbla.
<i>rwork</i>	(local). Workspace array of size <i>lwork</i> .
<i>lrwork</i>	(local) The size of the array <i>rwork</i> . $lrwork \geq 1 + 9*n + 3*np*nq,$

```
with np = numroc( n, nb, myrow, iarow, NPROW)
nq = numroc( n, nb, mycol, iacol, NPCOL)
```

iwork (local) Workspace array of size *liwork*.

liwork (local) , size of *iwork*.

```
liwork = 7*n + 8*npcol + 2.
```

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L'), or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

w (global).

Array of size *n*. If *info* = 0, *w* contains the eigenvalues in the ascending order.

z (local).

Array, global size $n*n$, local size *lld_z***LOCc*(*jz*+*n*-1).

The *z* parameter contains the orthonormal eigenvectors of the matrix *A*.

work[0] On exit, returns adequate workspace to allow optimal performance.

rwork[0] (local)

On output, *rwork*[0] returns workspace required to guarantee completion.

iwork[0] (local).

On return, *iwork*[0] contains the amount of integer workspace required.

info (global)

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*). If the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* > 0:

If *info* = 1 through *n*, the *i*-th eigenvalue did not converge.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?heevr

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using Relatively Robust Representation.

Syntax

```
void pcheevr(char* jobz, char* range, char* uplo, MKL_INT* n, MKL_Complex8* a, MKL_INT*
ia, MKL_INT* ja, MKL_INT* desca, float* vl, float* vu, MKL_INT* il, MKL_INT* iu,
MKL_INT* m, MKL_INT* nz, float* w, MKL_Complex8* z, MKL_INT* iz, MKL_INT* jz, MKL_INT*
descz, MKL_Complex8* work, MKL_INT* lwork, float* rwork, MKL_INT* lrwork, MKL_INT*
iwork, MKL_INT* liwork, MKL_INT* info);
```

```
void pzheevr(char* jobz, char* range, char* uplo, MKL_INT* n, MKL_Complex16* a, MKL_INT*
ia, MKL_INT* ja, MKL_INT* desca, double* vl, double* vu, MKL_INT* il, MKL_INT* iu,
MKL_INT* m, MKL_INT* nz, double* w, MKL_Complex16* z, MKL_INT* iz, MKL_INT* jz, MKL_INT*
descz, MKL_Complex16* work, MKL_INT* lwork, double* rwork, MKL_INT* lrwork, MKL_INT*
iwork, MKL_INT* liwork, MKL_INT* info);
```

Include Files

- `mkl_scalapack.h`

Description

`p?heevr` computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* distributed in 2D blockcyclic format by calling the recommended sequence of ScaLAPACK functions.

First, the matrix *A* is reduced to complex Hermitian tridiagonal form. Then, the eigenproblem is solved using the parallel MRRR algorithm. Last, if eigenvectors have been computed, a backtransformation is done.

Upon successful completion, each processor stores a copy of all computed eigenvalues in *w*. The eigenvector matrix *Z* is stored in 2D block-cyclic format distributed over all processors.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>jobz</i>	(global) Specifies whether or not to compute the eigenvectors: = 'N': Compute eigenvalues only. = 'V': Compute eigenvalues and eigenvectors.
<i>range</i>	(global) = 'A': all eigenvalues will be found. = 'V': all eigenvalues in the interval $[vl, vu]$ will be found. = 'I': the <i>il</i> -th through <i>iu</i> -th eigenvalues will be found.
<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	(global)

The number of rows and columns of the matrix A . $n \geq 0$

<i>a</i>	<p>Block-cyclic array, global size $n * n$, local size $lld_a * LOC_c(ja+n-1)$</p> <p>Contains the local pieces of the Hermitian distributed matrix A. If <i>uplo</i> = 'U', only the upper triangular part of <i>a</i> is used to define the elements of the Hermitian matrix. If <i>uplo</i> = 'L', only the lower triangular part of <i>a</i> is used to define the elements of the Hermitian matrix.</p>
<i>ia</i>	<p>(global)</p> <p>Global row index in the global matrix A that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.</p>
<i>ja</i>	<p>(global)</p> <p>Global column index in the global matrix A that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.</p>
<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>. (The ScaLAPACK descriptor length is <i>dlen_</i> = 9.)</p> <p>The array descriptor for the distributed matrix <i>a</i>. The descriptor stores details about the 2D block-cyclic storage, see the notes below. If <i>desca</i> is incorrect, <i>p?heevr</i> cannot work correctly.</p> <p>Also note the array alignment requirements specified below</p>
<i>vl</i>	<p>(global)</p> <p>If <i>range</i>='V', the lower bound of the interval to be searched for eigenvalues. Not referenced if <i>range</i> = 'A' or 'I'.</p>
<i>vu</i>	<p>(global)</p> <p>If <i>range</i>='V', the upper bound of the interval to be searched for eigenvalues. Not referenced if <i>range</i> = 'A' or 'I'.</p>
<i>il</i>	<p>(global)</p> <p>If <i>range</i>='I', the index (from smallest to largest) of the smallest eigenvalue to be returned. $il \geq 1$.</p> <p>Not referenced if <i>range</i> = 'A'.</p>
<i>iu</i>	<p>(global)</p> <p>If <i>range</i>='I', the index (from smallest to largest) of the largest eigenvalue to be returned. $\min(il, n) \leq iu \leq n$.</p> <p>Not referenced if <i>range</i> = 'A'.</p>
<i>iz</i>	<p>(global)</p> <p>Global row index in the global matrix Z that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.</p>
<i>jz</i>	<p>(global)</p>

Global column index in the global matrix Z that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

descz

(global and local) array of size *dlen_*.

The array descriptor for the distributed matrix z . *descz*[*ctxt_* - 1] must equal *desca*[*ctxt_* - 1]

work

(local workspace) array of size *lwork*

lwork

(local)

Size of *work* array, must be at least 3.

If only eigenvalues are requested:

$$lwork \geq n + \max(nb * (np00 + 1), nb * 3)$$

If eigenvectors are requested:

$$lwork \geq n + (np00 + mq00 + nb) * nb$$

For definitions of *np00* and *mq00*, see *lrwork*.

For optimal performance, greater workspace is needed, i.e.

$$lwork \geq \max(lwork, nhetrd_lwork)$$

Where *lwork* is as defined above, and

$$nhetrd_lwork = n + 2 * (anb + 1) * (4 * nps + 2) + (nps + 1) * nps$$

$$ictxt = desca[ctxt_ - 1]$$

$$anb = pjlavenv(ictxt, 3, 'PCHETTRD', 'L', 0, 0, 0, 0)$$

$$sqnpc = \text{sqrt}(\text{real}(nprow * npc0l))$$

$$nps = \max(\text{numroc}(n, 1, 0, 0, sqnpc), 2 * anb)$$

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

rwork

(local workspace) array of size *lrwork*

lrwork

(local)

Size of *rwork*, must be at least 3.

See below for definitions of variables used to define *lrwork*.

If no eigenvectors are requested (*jobz* = 'N') then

$$lrwork \geq 2 + 5 * n + \max(12 * n, nb * (np00 + 1))$$

If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required is:

$$lrwork \geq 2 + 5 * n + \max(18 * n, np00 * mq00 + 2 * nb * nb) +$$

$$(2 + \text{iceil}(neig, nprow * npc0l)) * n$$

NOTE

`iceil(x, y)` is the ceiling of x/y .

Variable definitions:

$neig$ = number of eigenvectors requested

$nb = desca[mb_ - 1] = desca[nb_ - 1] = descz[mb_ - 1] = descz[nb_ - 1]$

$nn = \max(n, nb, 2)$

$desca[rsrc_ - 1] = desca[csrc_ - 1] = descz[rsrc_ - 1] = descz[csrc_ - 1] = 0$

$np00 = \text{numroc}(nn, nb, 0, 0, nprow)$

$mq00 = \text{numroc}(\max(neig, nb, 2), nb, 0, 0, npcol)$

`iceil(x, y)` is a ScaLAPACK function returning $\text{ceiling}(x/y)$, and $nprow$ and $npcol$ can be determined by calling the function `blacs_gridinfo`.

If $lrwork = -1$, then $lrwork$ is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pxerbla`

$iwork$

(local workspace) array of size $liwork$

$liwork$

(local)

size of $iwork$

Let $nnp = \max(n, nprow * npcol + 1, 4)$. Then:

$liwork \geq 12 * nnp + 2 * n$ when the eigenvectors are desired

$liwork \geq 10 * nnp + 2 * n$ when only the eigenvalues have to be computed

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`

OUTPUT Parameters

a

The lower triangle (if $uplo='L'$) or the upper triangle (if $uplo='U'$) of a , including the diagonal, is destroyed.

m

(global)

Total number of eigenvalues found. $0 \leq m \leq n$.

nz

(global)

Total number of eigenvectors computed. $0 \leq nz \leq m$.

The number of columns of z that are filled.

If $jobz \neq 'V'$, nz is not referenced.

	If <i>jobz</i> = 'V', <i>nz</i> = <i>m</i>
<i>w</i>	(global) array of size <i>n</i> On normal exit, the first <i>m</i> entries contain the selected eigenvalues in ascending order.
<i>z</i>	(local) array, global size <i>n</i> * <i>n</i>), local size <i>lld_z</i> * <i>LOC_c</i> (<i>jz</i> + <i>n</i> -1) If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>work</i>	<i>work</i> [0] returns workspace adequate workspace to allow optimal performance.
<i>rwork</i>	On return, <i>rwork</i> [0] contains the optimal amount of workspace required for efficient execution. if <i>jobz</i> ='N' <i>rwork</i> [0] = optimal amount of workspace required to compute the eigenvalues. if <i>jobz</i> ='V' <i>rwork</i> [0] = optimal amount of workspace required to compute eigenvalues and eigenvectors.
<i>iwork</i>	On return, <i>iwork</i> [0] contains the amount of integer workspace required.
<i>info</i>	(global) = 0: successful exit < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Application Notes

The distributed submatrices *a*(*ia*:', *ja*:',) and *z*(*iz*:', *jz*:',) must satisfy the following alignment properties:

1. Identical (quadratic) dimension: *desca*[*m* - 1] = *descz*[*m* - 1] = *desca*[*n* - 1] = *descz*[*n* - 1]
2. Quadratic conformal blocking: *desca*[*mb* - 1] = *desca*[*nb* - 1] = *descz*[*mb* - 1] = *descz*[*nb* - 1],
desca[*rsrc* - 1] = *descz*[*rsrc* - 1]
3. mod(*ia*-1, *mb_a*) = mod(*iz*-1, *mb_z*) = 0

NOTE

mod(*x*, *y*) is the integer remainder of *x*/*y*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```
void pcheevx (char *jobz , char *range , char *uplo , MKL_INT *n , MKL_Complex8 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *vl , float *vu , MKL_INT *il ,
MKL_INT *iu , float *abstol , MKL_INT *m , MKL_INT *nz , float *w , float *orfac ,
MKL_Complex8 *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , MKL_Complex8 *work ,
MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT *info );
```

```
void pzheevx (char *jobz , char *range , char *uplo , MKL_INT *n , MKL_Complex16 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *vl , double *vu , MKL_INT *il ,
MKL_INT *iu , double *abstol , MKL_INT *m , MKL_INT *nz , double *w , double *orfac ,
MKL_Complex16 *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , MKL_Complex16 *work ,
MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , double *gap , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?heevx` function computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A by calling the recommended sequence of ScaLAPACK functions. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global) Must be 'N' or 'V'.

Specifies if it is necessary to compute the eigenvectors:

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

range (global) Must be 'A', 'V', or 'I'.

If *range* = 'A', all eigenvalues will be found.

If *range* = 'V', all eigenvalues in the half-open interval $[vl, vu]$ will be found.

If *range* = 'I', the eigenvalues with indices *il* through *iu* will be found.

uplo (global) Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

If *uplo* = 'U', *a* stores the upper triangular part of A .

	If <code>uplo = 'L'</code> , <code>a</code> stores the lower triangular part of <code>A</code> .
<code>n</code>	(global) The number of rows and columns of the matrix <code>A</code> ($n \geq 0$).
<code>a</code>	(local). Block cyclic array of global size $n*n$ and local size <code>lld_a*LOCc(ja+n-1)</code> . On entry, the Hermitian matrix <code>A</code> . If <code>uplo = 'U'</code> , only the upper triangular part of <code>A</code> is used to define the elements of the Hermitian matrix. If <code>uplo = 'L'</code> , only the lower triangular part of <code>A</code> is used to define the elements of the Hermitian matrix.
<code>ia, ja</code>	(global) The row and column indices in the global matrix <code>A</code> indicating the first row and the first column of the submatrix <code>A</code> , respectively.
<code>desca</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix <code>A</code> . If <code>desca[ctxt_ - 1]</code> is incorrect, <code>p?heevx</code> cannot guarantee correct error reporting.
<code>vl, vu</code>	(global) If <code>range = 'V'</code> , the lower and upper bounds of the interval to be searched for eigenvalues; not referenced if <code>range = 'A'</code> or <code>'I'</code> .
<code>il, iu</code>	(global) If <code>range = 'I'</code> , the indices of the smallest and largest eigenvalues to be returned. Constraints: $il \geq 1; \min(il, n) \leq iu \leq n$. Not referenced if <code>range = 'A'</code> or <code>'V'</code> .
<code>abstol</code>	(global). If <code>jobz='V'</code> , setting <code>abstol</code> to <code>p?lamch(context, 'U')</code> yields the most orthogonal eigenvectors. The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + eps * \max(a , b)$, where <code>eps</code> is the machine precision. If <code>abstol</code> is less than or equal to zero, then $eps * \text{norm}(T)$ will be used in its place, where <code>norm(T)</code> is the 1-norm of the tridiagonal matrix obtained by reducing <code>A</code> to tridiagonal form. Eigenvalues are computed most accurately when <code>abstol</code> is set to twice the underflow threshold $2 * p?lamch('S')$, not zero. If this function returns with $((\text{mod}(\text{info}, 2) \neq 0) . \text{or.} (\text{mod}(\text{info}/8, 2) \neq 0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting <code>abstol</code> to $2 * p?lamch('S')$.

NOTE

$\text{mod}(x, y)$ is the integer remainder of x/y .

`orfac` (global).

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol = orfac * \text{norm}(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of $1.0e-3$ is used if *orfac* is negative.

orfac should be identical on all processes.

iz, jz

(global) The row and column indices in the global matrix *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz

(global and local) array of size *dlen_*. The array descriptor for the distributed matrix *Z*. *descz[ctxt_ - 1]* must equal *desca[ctxt_ - 1]*.

work

(local).

Array of size *lwork*.

lwork

(local) The size of the array *work*.

If only eigenvalues are requested:

$$lwork \geq n + \max(nb * (np0 + 1), 3)$$

If eigenvectors are requested:

$$lwork \geq n + (np0 + mq0 + nb) * nb$$

with $nq0 = \text{numroc}(nn, nb, 0, 0, \text{NPCOL})$.

$$lwork \geq 5 * n + \max(5 * nn, np0 * mq0 + 2 * nb * nb) + \text{iceil}(\text{neig}, \text{NPROW} * \text{NPCOL}) * nn$$

For optimal performance, greater workspace is needed, that is

$$lwork \geq \max(lwork, \text{nhetrd_lwork})$$

where *lwork* is as defined above, and $\text{nhetrd_lwork} = n + 2 * (anb + 1) * (4 * nps + 2) + (nps + 1) * nps$

$$ictxt = \text{desca}[ctxt_ - 1]$$

$$anb = \text{pjlavenv}(ictxt, 3, 'pchettrd', 'L', 0, 0, 0, 0)$$

$$sqnpc = \text{sqrtd}(\text{double}(\text{NPROW} * \text{NPCOL}))$$

$$nps = \max(\text{numroc}(n, 1, 0, 0, sqnpc), 2 * anb)$$

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by *pxerbla*.

rwork

(local)

Workspace array of size *lwork*.

lrwork

(local) The size of the array *work*.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then $lrwork \geq 5 * nn + 4 * n$.

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

$$lwork \geq 4*n + \max(5*nn, np0*mq0+2*nb*nb) + \text{iceil}(neig, \text{NPROW}*NPCOL)*nn$$

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following values to *lwork*:

$$(clustersize-1)*n,$$

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

$$\{w[k-1], \dots, w[k+clustersize-2] \mid w[j] \leq w[j-1] + orfac*2*norm(A)\}.$$

Variable definitions:

neig = number of eigenvectors requested;

nb = *desca*[*mb_* - 1] = *desca*[*nb_* - 1] = *descz*[*mb_* - 1] = *descz*[*nb_* - 1];

nn = max(*n*, *NB*, 2);

desca[*rsrc_* - 1] = *desca*[*nb_* - 1] = *descz*[*rsrc_* - 1] = *descz*[*csrc_* - 1] = 0;

np0 = numroc(*nn*, *nb*, 0, 0, *NPROW*);

mq0 = numroc(max(*neig*, *nb*, 2), *nb*, 0, 0, *NPCOL*);

iceil(*x*, *y*) is a ScaLAPACK function returning ceiling(*x*/*y*)

When *lwork* is too small:

If *lwork* is too small to guarantee orthogonality, *p?heevx* attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues. If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned. Note that when *range*='V', *p?heevx* does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow *p?heevx* to compute the eigenvalues, *p?heevx* will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality and performance:

If *clustersize* $\geq n/\text{sqrt}(\text{NPROW}*NPCOL)$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1) *p?stein* will perform no better than *?stein* on 1 processor.

For *clustersize* = *n*/*sqrt*(*NPROW***NPCOL*) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize* > *n*/*sqrt*(*NPROW***NPCOL*) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pxerbla`.

$iwork$ (local) Workspace array.

$liwork$ (local), size of $iwork$.

$liwork \geq 6 * nnp$

Where: $nnp = \max(n, NPROW * NPCOL + 1, 4)$

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a On exit, the lower triangle (if $uplo = 'L'$), or the upper triangle (if $uplo = 'U'$) of A , including the diagonal, is overwritten.

m (global) The total number of eigenvalues found; $0 \leq m \leq n$.

nz (global) Total number of eigenvectors computed. $0 \leq nz \leq m$.

The number of columns of z that are filled.

If $jobz \neq 'V'$, nz is not referenced.

If $jobz = 'V'$, $nz = m$ unless the user supplies insufficient space and `p?heevx` is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in z ($m \leq descz[n_ - 1]$) and sufficient workspace to compute them. (See $lwork$). `p?heevx` is always able to detect insufficient space without computation unless $range = 'V'$.

w (global).

Array of size n . The first m elements contain the selected eigenvalues in ascending order.

z (local).

Array, global size $n * n$, local size $lld_z * LOCc(jz + n - 1)$.

If $jobz = 'V'$, then on normal exit the first m columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced.

$work[0]$ On exit, returns adequate workspace to allow optimal performance.

$rwork$ (local).

Array of size *lwork*. On return, *rwork*[0] contains the optimal amount of workspace required for efficient execution.

If *jobz*='N' *rwork*[0] = optimal amount of workspace required to compute eigenvalues efficiently.

If *jobz*='V' *rwork*[0] = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If *range*='V', it is assumed that all eigenvectors may be required.

iwork[0]

(local)

On return, *iwork*[0] contains the amount of integer workspace required.

ifail

(global)

Array of size *n*.

If *jobz* = 'V', then on normal exit, the first *m* elements of *ifail* are zero. If $(\text{mod}(\text{info}, 2) \neq 0)$ on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

iclustr

(global)

Array of size $2 \times \text{NPROW} \times \text{NPCOL}$.

This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*[2**i* - 2] to *iclustr*[2**i* - 1], could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal.

iclustr is a zero terminated array. (*iclustr*[2**k* - 1] ≠ 0 and *iclustr*[2**k*]=0) if and only if *k* is the number of clusters. *iclustr* is not referenced if *jobz* = 'N'.

gap

(global)

Array of size $(\text{NPROW} \times \text{NPCOL})$

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as $(C \times n) / \text{gap}(i)$ where *C* is a small constant.

info

(global)

If *info* = 0, the execution is successful.

If *info* < 0:

If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*). If the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* > 0:

If $(\text{mod}(\text{info}, 2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure *abstol*= $2.0 \times \text{p2lamch}('U')$

If $(\text{mod}(\text{info}/2, 2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(\text{info}/4, 2) \neq 0)$, then space limit prevented [p?syevx](#) from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If $(\text{mod}(\text{info}/8, 2) \neq 0)$, then [p?stebz](#) failed to compute eigenvalues. Ensure $\text{abstol} = 2.0 * \text{p?lamch}('U')$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gesvd

Computes the singular value decomposition of a general matrix, optionally computing the left and/or right singular vectors.

Syntax

```
void psgesvd (char *jobu , char *jobvt , MKL_INT *m , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *s , float *u , MKL_INT *iu , MKL_INT *ju ,
MKL_INT *descu , float *vt , MKL_INT *ivt , MKL_INT *jvt , MKL_INT *descvt , float
*work , MKL_INT *lwork , float *rwork , MKL_INT *info );

void pdgesvd (char *jobu , char *jobvt , MKL_INT *m , MKL_INT *n , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *s , double *u , MKL_INT *iu , MKL_INT *ju ,
MKL_INT *descu , double *vt , MKL_INT *ivt , MKL_INT *jvt , MKL_INT *descvt , double
*work , MKL_INT *lwork , double *rwork , MKL_INT *info );

void pcgesvd (char *jobu , char *jobvt , MKL_INT *m , MKL_INT *n , MKL_Complex8 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *s , MKL_Complex8 *u , MKL_INT *iu ,
MKL_INT *ju , MKL_INT *descu , MKL_Complex8 *vt , MKL_INT *ivt , MKL_INT *jvt , MKL_INT
*descvt , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *info );

void pzgesvd (char *jobu , char *jobvt , MKL_INT *m , MKL_INT *n , MKL_Complex16 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *s , MKL_Complex16 *u , MKL_INT
*iu , MKL_INT *ju , MKL_INT *descu , MKL_Complex16 *vt , MKL_INT *ivt , MKL_INT *jvt ,
MKL_INT *descvt , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT
*info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?gesvd` function computes the singular value decomposition (SVD) of an m -by- n matrix A , optionally computing the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^T,$$

where Σ is an m -by- n matrix that is zero except for its $\min(m, n)$ diagonal elements, U is an m -by- m orthogonal matrix, and V is an n -by- n orthogonal matrix. The diagonal elements of Σ are the singular values of A and the columns of U and V are the corresponding right and left singular vectors, respectively. The singular values are returned in array *s* in decreasing order and only the first $\min(m, n)$ columns of U and rows of $vt = V^T$ are computed.

Product and Performance Information

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

Notice revision #20201201

NOTE

The distributed submatrix $\text{sub}(A)$ must verify certain alignment properties. These expressions must be true:

- $mb_a = nb_a = nb$
- $iroffa = icoffa$

where:

- $iroffa = \text{mod}(ia-1, nb)$
- $icoffa = \text{mod}(ja-1, nb)$

Input Parameters

mp = number of local rows in A and U

nq = number of local columns in A and VT

$size = \min(m, n)$

$sizeq$ = number of local columns in U

$sizep$ = number of local rows in VT

$jobu$	(global) Specifies options for computing all or part of the matrix U . If $jobu = 'V'$, the first $size$ columns of U (the left singular vectors) are returned in the array u ; If $jobu = 'N'$, no columns of U (no left singular vectors) are computed.
$jobvt$	(global) Specifies options for computing all or part of the matrix V^T . If $jobvt = 'V'$, the first $size$ rows of V^T (the right singular vectors) are returned in the array vt ; If $jobvt = 'N'$, no rows of V^T (no right singular vectors) are computed.
m	(global) The number of rows of the matrix A ($m \geq 0$).
n	(global) The number of columns in A ($n \geq 0$).
a	(local). Block cyclic array, global size (m, n) , local size (mp, nq) .
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

<i>iu, ju</i>	(global) The row and column indices in the global matrix <i>U</i> indicating the first row and the first column of the submatrix <i>U</i> , respectively.
<i>descu</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>U</i> .
<i>ivt, jvt</i>	(global) The row and column indices in the global matrix <i>VT</i> indicating the first row and the first column of the submatrix <i>VT</i> , respectively.
<i>descvt</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>VT</i> .
<i>work</i>	(local). Workspace array of size <i>lwork</i>
<i>lwork</i>	(local) The size of the array <i>work</i> ; $lwork > 2 + 6 * sizeb + \max(watobd, wbdto svd),$ where $sizeb = \max(m, n)$, and <i>watobd</i> and <i>wbdto svd</i> refer, respectively, to the workspace required to bidiagonalize the matrix <i>A</i> and to go from the bidiagonal matrix to the singular value decomposition <i>USVT</i> . For <i>watobd</i> , the following holds: $watobd = \max(\max(wp?lange, wp?gebrd), \max(wp?lared2d, wp?lared1d)),$ where <i>wp?lange</i> , <i>wp?lared1d</i> , <i>wp?lared2d</i> , <i>wp?gebrd</i> are the workspaces required respectively for the subprograms <i>p?lange</i> , <i>p?lared1d</i> , <i>p?lared2d</i> , <i>p?gebrd</i> . Using the standard notation $mp = \text{numroc}(m, mb, MYROW, desca[ctxt_ - 1], NPROW),$ $nq = \text{numroc}(n, nb, MYCOL, desca[lld_ - 1], NPCOL),$ the workspaces required for the above subprograms are $wp?lange = mp,$ $wp?lared1d = nq0,$ $wp?lared2d = mp0,$ $wp?gebrd = nb * (mp + nq + 1) + nq,$ where <i>nq0</i> and <i>mp0</i> refer, respectively, to the values obtained at <i>MYCOL</i> = 0 and <i>MYROW</i> = 0. In general, the upper limit for the workspace is given by a workspace required on processor (0,0): $watobd \leq nb * (mp0 + nq0 + 1) + nq0.$ In case of a homogeneous process grid this upper limit can be used as an estimate of the minimum workspace for every processor. For <i>wbdto svd</i> , the following holds: $wbdto svd = size * (wantu * nru + wantvt * ncvt) + \max(w?bdsqr, \max(wantu * wp?ormbrqln, wantvt * wp?ormbrprt)),$ where

$wantu(wantvt) = 1$, if left/right singular vectors are wanted, and $wantu(wantvt) = 0$, otherwise. $w?bdsqr$, $wp?ormbrqln$, and $wp?ormbrprt$ refer respectively to the workspace required for the subprograms $?bdsqr$, $p?ormbr(qln)$, and $p?ormbr(prt)$, where qln and prt are the values of the arguments *vect*, *side*, and *trans* in the call to $p?ormbr$. nru is equal to the local number of rows of the matrix U when distributed 1-dimensional "column" of processes. Analogously, $ncvt$ is equal to the local number of columns of the matrix VT when distributed across 1-dimensional "row" of processes. Calling the LAPACK procedure $?bdsqr$ requires

$$w?bdsqr = \max(1, 2*size + (2*size - 4) * \max(wantu, wantvt))$$

on every processor. Finally,

$$wp?ormbrqln = \max((nb*(nb-1))/2, (sizeq+mp)*nb)+nb*nb,$$

$$wp?ormbrprt = \max((mb*(mb-1))/2, (sizep+nq)*mb)+mb*mb,$$

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum size for the work array. The required workspace is returned as the first element of *work* and no error message is issued by $pxerbla$.

rwork Workspace array of size $1 + 4*sizeb$. Not used for $psgesvd$ and $pdgesvd$.

Output Parameters

a On exit, the contents of *a* are destroyed.

s (global).
Array of size *size*.
Contains the singular values of A sorted so that $s(i) \geq s(i+1)$.

u (local).
local size $mp*sizeq$, global size $m*size$)
If $jobu = 'V'$, *u* contains the first $\min(m, n)$ columns of U .
If $jobu = 'N'$ or $'O'$, *u* is not referenced.

vt (local).
local size (*sizep*, *nq*), global size (*size*, *n*)
If $jobvt = 'V'$, *vt* contains the first *size* rows of V^T if $jobu = 'N'$, *vt* is not referenced.

work On exit, if *info* = 0, then *work*[0] returns the required minimal size of *lwork*.

rwork On exit, if *info* = 0, then *rwork*[0] returns the required size of *rwork*.

info (global)
If *info* = 0, the execution is successful.
If *info* < 0, If *info* = -*i*, the *i*th parameter had an illegal value.
If *info* > 0 *i*, then if $?bdsqr$ did not converge,

If $info = \min(m, n) + 1$, then `p?gesvd` has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from `p?gesvd` cannot be guaranteed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```
void pssygvx (MKL_INT *ibtype , char *jobz , char *range , char *uplo , MKL_INT *n ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT
*jb , MKL_INT *descb , float *vl , float *vu , MKL_INT *il , MKL_INT *iu , float
*abstol , MKL_INT *m , MKL_INT *nz , float *w , float *orfac , float *z , MKL_INT *iz ,
MKL_INT *jz , MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT
*liwork , MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT *info );

void pdsygvx (MKL_INT *ibtype , char *jobz , char *range , char *uplo , MKL_INT *n ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , double *vl , double *vu , MKL_INT *il , MKL_INT *iu ,
double *abstol , MKL_INT *m , MKL_INT *nz , double *w , double *orfac , double *z ,
MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , double *work , MKL_INT *lwork , MKL_INT
*iwork , MKL_INT *liwork , MKL_INT *ifail , MKL_INT *iclustr , double *gap , MKL_INT
*info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?sygvx` function computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$\text{sub}(A) * x = \lambda * \text{sub}(B) * x$, $\text{sub}(A) \text{ sub}(B) * x = \lambda * x$, or $\text{sub}(B) * \text{sub}(A) * x = \lambda * x$.

Here x denotes eigen vectors, λ (*lambda*) denotes eigenvalues, $\text{sub}(A)$ denoting $A(ia:ia+n-1, ja:ja+n-1)$ is assumed to symmetric, and $\text{sub}(B)$ denoting $B(ib:ib+n-1, jb:jb+n-1)$ is also positive definite.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

ibtype (global) Must be 1 or 2 or 3.
Specifies the problem type to be solved:
If *ibtype* = 1, the problem type is $\text{sub}(A) * x = \text{lambda} * \text{sub}(B) * x$;
If *ibtype* = 2, the problem type is $\text{sub}(A) * \text{sub}(B) * x = \text{lambda} * x$;

	<p>If $ibtype = 3$, the problem type is $\text{sub}(B) * \text{sub}(A) * x = \text{lambda} * x$.</p>
<i>jobz</i>	<p>(global) Must be 'N' or 'V'.</p> <p>If $jobz = 'N'$, then compute eigenvalues only.</p> <p>If $jobz = 'V'$, then compute eigenvalues and eigenvectors.</p>
<i>range</i>	<p>(global) Must be 'A' or 'V' or 'I'.</p> <p>If $range = 'A'$, the function computes all eigenvalues.</p> <p>If $range = 'V'$, the function computes eigenvalues in the interval: $[vl, vu]$</p> <p>If $range = 'I'$, the function computes eigenvalues with indices il through iu.</p>
<i>uplo</i>	<p>(global) Must be 'U' or 'L'.</p> <p>If $uplo = 'U'$, arrays a and b store the upper triangles of $\text{sub}(A)$ and $\text{sub}(B)$;</p> <p>If $uplo = 'L'$, arrays a and b store the lower triangles of $\text{sub}(A)$ and $\text{sub}(B)$.</p>
<i>n</i>	(global) The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$, $n \geq 0$.
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$. On entry, this array contains the local pieces of the n-by-n symmetric distributed matrix $\text{sub}(A)$.</p> <p>If $uplo = 'U'$, the leading n-by-n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix.</p> <p>If $uplo = 'L'$, the leading n-by-n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix.</p>
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A , respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A . If $desca[ctxt_ - 1]$ is incorrect, $p?sygvx$ cannot guarantee correct error reporting.
<i>b</i>	<p>(local).</p> <p>Pointer into the local memory to an array of size $lld_b * LOCC(jb+n-1)$. On entry, this array contains the local pieces of the n-by-n symmetric distributed matrix $\text{sub}(B)$.</p> <p>If $uplo = 'U'$, the leading n-by-n upper triangular part of $\text{sub}(B)$ contains the upper triangular part of the matrix.</p> <p>If $uplo = 'L'$, the leading n-by-n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix.</p>
<i>ib, jb</i>	(global) The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B , respectively.

<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> . <i>descb[ctxt_ - 1]</i> must be equal to <i>desca[ctxt_ - 1]</i> .
<i>vl, vu</i>	(global) If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	(global) If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $il \geq 1, \min(il, n) \leq iu \leq n$ If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	(global) If <i>jobz</i> ='V', setting <i>abstol</i> to <code>p?lamch(context, 'U')</code> yields the most orthogonal eigenvectors. The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + eps * \max(a , b),$ where <i>eps</i> is the machine precision. If <i>abstol</i> is less than or equal to zero, then <i>eps</i> * <code>norm(T)</code> will be used in its place, where <code>norm(T)</code> is the 1-norm of the tridiagonal matrix obtained by reducing <i>A</i> to tridiagonal form. Eigenvalues will be computed most accurately when <i>abstol</i> is set to twice the underflow threshold <code>2*p?lamch('S')</code> not zero. If this function returns with <code>((mod(info,2)≠0) or (mod(info/8,2)≠0))</code> , indicating that some eigenvalues or eigenvectors did not converge, try setting <i>abstol</i> to <code>2*p?lamch('S')</code> .
<hr/> NOTE <code>mod(x, y)</code> is the integer remainder of <i>x/y</i> . <hr/>	
<i>orfac</i>	(global). Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol = orfac * \text{norm}(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see <i>lwork</i>), <i>tol</i> may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if <i>orfac</i> equals zero. A default value of 1.0e-3 is used if <i>orfac</i> is negative. <i>orfac</i> should be identical on all processes.
<i>iz, jz</i>	(global) The row and column indices in the global matrix <i>Z</i> indicating the first row and the first column of the submatrix <i>Z</i> , respectively.
<i>descz</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> . <i>descz[ctxt_ - 1]</i> must equal <i>desca[ctxt_ - 1]</i> .
<i>work</i>	(local)

lwork

Workspace array of size *lwork*

(local)

Size of the array *work*. See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then $lwork \geq 5*n + \max(5*nn, NB*(np0 + 1))$.

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

$lwork \geq 5*n + \max(5*nn, np0*mq0 + 2*nb*nb) + \text{iceil}(neig, NPROW*NPCOL)*nn$.

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality at the cost of potentially poor performance you should add the following to *lwork*:

$(clustersize-1)*n$,

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

$\{w[k-1], \dots, w[k+clustersize-2] | w[j] \leq w[j-1] + orfac*2*norm(A)\}$

Variable definitions:

neig = number of eigenvectors requested,

$nb = desca[mb_ - 1] = desca[nb_ - 1] = descz[mb_ - 1] = descz[nb_ - 1]$,

$nn = \max(n, nb, 2)$,

$desca[rsrc_ - 1] = desca[nb_ - 1] = descz[rsrc_ - 1] = descz[csrc_ - 1] = 0$,

$np0 = \text{numroc}(nn, nb, 0, 0, NPROW)$,

$mq0 = \text{numroc}(\max(neig, nb, 2), nb, 0, 0, NPCOL)$

iceil(*x*, *y*) is a ScaLAPACK function returning $\text{ceiling}(x/y)$

If *lwork* is too small to guarantee orthogonality, *p?syevx* attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, *p?sygvx* computes the eigenvalues and as many eigenvectors as possible.

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

$lwork \geq \max(lwork, 5*n + nsytrd_lwopt, nsygst_lwopt)$, where

lwork, as defined previously, depends upon the number of eigenvectors requested, and

```
nsytrd_lwopt = n + 2*(anb+1)*(4*nps+2) + (nps+3)*nps
nsygst_lwopt = 2*np0*nb + nq0*nb + nb*nb
anb = pjslaenv(desca[ctxt_ - 1], 3, p?sytrd, 'L', 0, 0, 0, 0)
sqnpc = int(sqrt(dble(NPROW * NPCOL)))
nps = max(numroc(n, 1, 0, 0, sqnpc), 2*anb)
NB = desca[mb_ - 1]
np0 = numroc(n, nb, 0, 0, NPROW)
nq0 = numroc(n, nb, 0, 0, NPCOL)
```

numroc is a ScaLAPACK tool functions;

pjslaenv is a ScaLAPACK environmental inquiry function

MYROW, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the function *blacs_gridinfo*.

For large *n*, no extra workspace is needed, however the biggest boost in performance comes for small *n*, so it is wise to provide the extra workspace (typically less than a Megabyte per process).

If *clustersize* $\geq n/\sqrt{NPROW \cdot NPCOL}$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, *clustersize* = *n*-1) *p?stein* will perform no better than *?stein* on a single processor.

For *clustersize* = $n/\sqrt{NPROW \cdot NPCOL}$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize* $> n/\sqrt{NPROW \cdot NPCOL}$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by *pserbla*.

iwork

(local) Workspace array.

liwork

(local) , size of *iwork*.

$liwork \geq 6 \cdot nnp$

Where:

$nnp = \max(n, NPROW \cdot NPCOL + 1, 4)$

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pserbla*.

Output Parameters

<i>a</i>	<p>On exit,</p> <p>If <i>jobz</i> = 'V', and if <i>info</i> = 0, sub(A) contains the distributed matrix Z of eigenvectors. The eigenvectors are normalized as follows:</p> <p>for <i>ibtype</i> = 1 or 2, $Z^T \cdot \text{sub}(B) \cdot Z = I$;</p> <p>for <i>ibtype</i> = 3, $Z^T \cdot \text{inv}(\text{sub}(B)) \cdot Z = I$.</p> <p>If <i>jobz</i> = 'N', then on exit the upper triangle (if <i>uplo</i>='U') or the lower triangle (if <i>uplo</i>='L') of sub(A), including the diagonal, is destroyed.</p>
<i>b</i>	<p>On exit, if <i>info</i> ≤ <i>n</i>, the part of sub(B) containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $\text{sub}(B) = U^T \cdot U$ or $\text{sub}(B) = L \cdot L^T$.</p>
<i>m</i>	(global) The total number of eigenvalues found, $0 \leq m \leq n$.
<i>nz</i>	<p>(global)</p> <p>Total number of eigenvectors computed. $0 \leq nz \leq m$. The number of columns of z that are filled.</p> <p>If <i>jobz</i> ≠ 'V', <i>nz</i> is not referenced.</p> <p>If <i>jobz</i> = 'V', <i>nz</i> = <i>m</i> unless the user supplies insufficient space and p?sygvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in <i>z</i> ($m \leq \text{descz}(n_)$) and sufficient workspace to compute them. (See <i>lwork</i> below.) p?sygvx is always able to detect insufficient space without computation unless <i>range</i>='V'.</p>
<i>w</i>	<p>(global)</p> <p>Array of size <i>n</i>. On normal exit, the first <i>m</i> entries contain the selected eigenvalues in ascending order.</p>
<i>z</i>	<p>(local).</p> <p>global size $n \cdot n$, local size $\text{lld_z} \cdot \text{LOCc}(jz+n-1)$.</p> <p>If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i>.</p> <p>If <i>jobz</i> = 'N', then z is not referenced.</p>
<i>work</i>	<p>If <i>jobz</i>='N' <i>work</i>[0] = optimal amount of workspace required to compute eigenvalues efficiently</p> <p>If <i>jobz</i> = 'V' <i>work</i>[0] = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.</p> <p>If <i>range</i>='V', it is assumed that all eigenvectors may be required.</p>
<i>ifail</i>	<p>(global)</p> <p>Array of size <i>n</i>.</p>

ifail provides additional information when *info* ≠ 0

If $(\text{mod}(\text{info}/16, 2) \neq 0)$ then *ifail*[0] indicates the order of the smallest minor which is not positive definite. If $(\text{mod}(\text{info}, 2) \neq 0)$ on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.

If neither of the above error conditions hold and *jobz* = 'V', then the first *m* elements of *ifail* are set to zero.

iclustr

(global)

Array of size $(2 * \text{NPROW} * \text{NPCOL})$. This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*[2**i* - 2] to *iclustr*[2**i* - 1], could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr* is a zero terminated array.

$(\text{iclustr}[2*k - 1] \neq 0 \text{ and } \text{iclustr}[2*k] = 0)$ if and only if *k* is the number of clusters *iclustr* is not referenced if *jobz* = 'N'.

gap

(global)

Array of size $\text{NPROW} * \text{NPCOL}$. This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as $(C * n) / \text{gap}[i - 1]$, where *C* is a small constant.

info

(global)

If *info* = 0, the execution is successful.

If *info* < 0: the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i * 100 + j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

If *info* > 0:

If $(\text{mod}(\text{info}, 2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.

If $(\text{mod}(\text{info}, 2, 2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(\text{info}/4, 2) \neq 0)$, then space limit prevented *p?sygvx* from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If $(\text{mod}(\text{info}/8, 2) \neq 0)$, then *p?stebz* failed to compute eigenvalues.

If $(\text{mod}(\text{info}/16, 2) \neq 0)$, then *B* was not positive definite. *ifail*(1) indicates the order of the smallest minor which is not positive definite.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

```
void pchegvx (MKL_INT *ibtype , char *jobz , char *range , char *uplo , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *vl , float *vu , MKL_INT *il ,
MKL_INT *iu , float *abstol , MKL_INT *m , MKL_INT *nz , float *w , float *orfac ,
MKL_Complex8 *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , MKL_Complex8 *work ,
MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT *info );
```

```
void pzhegvx (MKL_INT *ibtype , char *jobz , char *range , char *uplo , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *vl , double *vu , MKL_INT *il ,
MKL_INT *iu , double *abstol , MKL_INT *m , MKL_INT *nz , double *w , double *orfac ,
MKL_Complex16 *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , MKL_Complex16 *work ,
MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , double *gap , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?hegvx function computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x, \quad \text{sub}(A) * \text{sub}(B) * x = \lambda * x, \quad \text{or} \quad \text{sub}(B) * \text{sub}(A) * x = \lambda * x.$$

Here sub(A) denoting $A(ia:ia+n-1, ja:ja+n-1)$ and sub(B) are assumed to be Hermitian and sub(B) denoting $B(ib:ib+n-1, jb:jb+n-1)$ is also positive definite.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

ibtype

(global) Must be 1 or 2 or 3.

Specifies the problem type to be solved:

If *ibtype* = 1, the problem type is

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x;$$

If *ibtype* = 2, the problem type is

$$\text{sub}(A) * \text{sub}(B) * x = \lambda * x;$$

If *ibtype* = 3, the problem type is

$$\text{sub}(B) * \text{sub}(A) * x = \lambda * x.$$

<i>jobz</i>	<p>(global) Must be 'N' or 'V'.</p> <p>If <i>jobz</i> = 'N', then compute eigenvalues only.</p> <p>If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.</p>
<i>range</i>	<p>(global) Must be 'A' or 'V' or 'I'.</p> <p>If <i>range</i> = 'A', the function computes all eigenvalues.</p> <p>If <i>range</i> = 'V', the function computes eigenvalues in the interval: [<i>vl</i>, <i>vu</i>]</p> <p>If <i>range</i> = 'I', the function computes eigenvalues with indices <i>il</i> through <i>iu</i>.</p>
<i>uplo</i>	<p>(global) Must be 'U' or 'L'.</p> <p>If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of sub(<i>A</i>) and sub(<i>B</i>);</p> <p>If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of sub(<i>A</i>) and sub(<i>B</i>).</p>
<i>n</i>	<p>(global)</p> <p>The order of the matrices sub(<i>A</i>) and sub(<i>B</i>) ($n \geq 0$).</p>
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size <code>lld_a*LOCc(ja+n-1)</code>. On entry, this array contains the local pieces of the <i>n</i>-by-<i>n</i> Hermitian distributed matrix sub(<i>A</i>). If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of sub(<i>A</i>) contains the upper triangular part of the matrix. If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular part of the matrix.</p>
<i>ia, ja</i>	<p>(global)</p> <p>The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of the submatrix <i>A</i>, respectively.</p>
<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>.</p> <p>The array descriptor for the distributed matrix <i>A</i>. If <i>desca</i>[<i>ctxt_</i> - 1] is incorrect, p?hegvx cannot guarantee correct error reporting.</p>
<i>b</i>	<p>(local).</p> <p>Pointer into the local memory to an array of size <code>lld_b*LOCc(jb+n-1)</code>. On entry, this array contains the local pieces of the <i>n</i>-by-<i>n</i> Hermitian distributed matrix sub(<i>B</i>).</p> <p>If <i>uplo</i> = 'U', the leading <i>n</i>-by-<i>n</i> upper triangular part of sub(<i>B</i>) contains the upper triangular part of the matrix.</p> <p>If <i>uplo</i> = 'L', the leading <i>n</i>-by-<i>n</i> lower triangular part of sub(<i>B</i>) contains the lower triangular part of the matrix.</p>
<i>ib, jb</i>	<p>(global)</p> <p>The row and column indices in the global matrix <i>B</i> indicating the first row and the first column of the submatrix <i>B</i>, respectively.</p>

<i>descb</i>	<p>(global and local) array of size <i>dlen_</i>.</p> <p>The array descriptor for the distributed matrix <i>B.descb[ctxt_ - 1]</i> must be equal to <i>desca[ctxt_ - 1]</i>.</p>
<i>vl, vu</i>	<p>(global)</p> <p>If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.</p> <p>If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.</p>
<i>il, iu</i>	<p>(global)</p> <p>If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $il \geq 1, \min(il, n) \leq iu \leq n$</p> <p>If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.</p>
<i>abstol</i>	<p>(global)</p> <p>If <i>jobz</i>='V', setting <i>abstol</i> to <code>p?lamch(context, 'U')</code> yields the most orthogonal eigenvectors.</p> <p>The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to</p> $abstol + eps * \max(a , b),$ <p>where <i>eps</i> is the machine precision. If <i>abstol</i> is less than or equal to zero, then $eps * \text{norm}(\mathbb{T})$ will be used in its place, where $\text{norm}(\mathbb{T})$ is the 1-norm of the tridiagonal matrix obtained by reducing <i>A</i> to tridiagonal form.</p> <p>Eigenvalues will be computed most accurately when <i>abstol</i> is set to twice the underflow threshold $2 * p?lamch('S')$ not zero. If this function returns with $((\text{mod}(\text{info}, 2) \neq 0) .or. * (\text{mod}(\text{info}/8, 2) \neq 0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting <i>abstol</i> to $2 * p?lamch('S')$.</p>
<hr/> <p>NOTE</p> <p>$\text{mod}(x, y)$ is the integer remainder of x/y.</p> <hr/>	
<i>orfac</i>	<p>(global).</p> <p>Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $\text{tol} = \text{orfac} * \text{norm}(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see <i>lwork</i>), <i>tol</i> may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if <i>orfac</i> equals zero. A default value of 1.0E-3 is used if <i>orfac</i> is negative. <i>orfac</i> should be identical on all processes.</p>
<i>iz, jz</i>	<p>(global) The row and column indices in the global matrix <i>Z</i> indicating the first row and the first column of the submatrix <i>Z</i>, respectively.</p>
<i>descz</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>Z.descz[ctxt_ - 1]</i> must equal <i>desca[ctxt_ - 1]</i>.</p>

<i>work</i>	(local) Workspace array of size <i>lwork</i>
<i>lwork</i>	(local). The size of the array <i>work</i> . If only eigenvalues are requested: $lwork \geq n + \max(NB * (np0 + 1), 3)$ If eigenvectors are requested: $lwork \geq n + (np0 + mq0 + NB) * NB$ with $nq0 = \text{numroc}(nn, NB, 0, 0, NPCOL)$. For optimal performance, greater workspace is needed, that is $lwork \geq \max(lwork, n, nhetr_lwopt, nhegst_lwopt)$ where <i>lwork</i> is as defined above, and $nhetr_lwork = 2 * (anb + 1) * (4 * nps + 2) + (nps + 1) * nps;$ $nhegst_lwopt = 2 * np0 * nb + nq0 * nb + nb * nb$ $nb = \text{desca}[mb_ - 1]$ $np0 = \text{numroc}(n, nb, 0, 0, NPROW)$ $nq0 = \text{numroc}(n, nb, 0, 0, NPCOL)$ $ictxt = \text{desca}[ctxt_ - 1]$ $anb = \text{pjlaenv}(ictxt, 3, 'p?hettrd', 'L', 0, 0, 0, 0)$ $sqnpc = \text{sqrt}(\text{dble}(NPROW * NPCOL))$ $nps = \max(\text{numroc}(n, 1, 0, 0, sqnpc), 2 * anb)$ numroc is a ScaLAPACK tool functions; pjlaenv is a ScaLAPACK environmental inquiry function MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo. If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxebla.
<i>rwork</i>	(local) Workspace array of size <i>lrwork</i> .
<i>lrwork</i>	(local) The size of the array <i>rwork</i> . See below for definitions of variables used to define <i>lrwork</i> . If no eigenvectors are requested (<i>jobz</i> = 'N'), then $lrwork \geq 5 * nn + 4 * n$ If eigenvectors are requested (<i>jobz</i> = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is: $lrwork \geq 4 * n + \max(5 * nn, np0 * mq0) + \text{iceil}(neig, NPROW * NPCOL) * nn$

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following value to *lwork*:

```
(clustersize-1)*n,
```

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

```
{w[k - 1], ..., w[k+clustersize - 2] | w[j] ≤ w[j - 1] + orfac*2*norm(A) }
```

Variable definitions:

neig = number of eigenvectors requested;

```
nb = desca[mb_ - 1] = desca[nb_ - 1] = descz[mb_ - 1] = descz[nb_ - 1];
```

```
nn = max(n, nb, 2);
```

```
desca[rsrc_ - 1] = desca[nb_ - 1] = descz[rsrc_ - 1] = descz[csrc_ - 1] = 0;
```

```
np0 = numroc(nn, nb, 0, 0, NPROW);
```

```
mq0 = numroc(max(neig, nb, 2), nb, 0, 0, NPCOL);
```

iceil(x, y) is a ScaLAPACK function returning ceiling(*x/y*).

When *lwork* is too small:

If *lwork* is too small to guarantee orthogonality, *p?hegvx* attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -25 is returned. Note that when *range*='V', *p?hegvx* does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow *p?hegvx* to compute the eigenvalues, *p?hegvx* will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality & performance:

If $clustersize > n/\sqrt{NPROW \cdot NPCOL}$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, $clustersize = n-1$) *p?stein* will perform no better than *?stein* on 1 processor.

For $clustersize = n/\sqrt{NPROW \cdot NPCOL}$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For $clustersize > n/\sqrt{NPROW \cdot NPCOL}$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pxerbla`.

$iwork$ (local) Workspace array.

$liwork$ (local) , size of $iwork$.

$liwork \geq 6 * nnp$

Where: $nnp = \max(n, NPROW * NPCOL + 1, 4)$

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a On exit, if $jobz = 'V'$, then if $info = 0$, $\text{sub}(A)$ contains the distributed matrix Z of eigenvectors.

The eigenvectors are normalized as follows:

If $ibtype = 1$ or 2 , then $Z^H * \text{sub}(B) * Z = I$;

If $ibtype = 3$, then $Z^H * \text{inv}(\text{sub}(B)) * Z = I$.

If $jobz = 'N'$, then on exit the upper triangle (if $uplo='U'$) or the lower triangle (if $uplo='L'$) of $\text{sub}(A)$, including the diagonal, is destroyed.

b On exit, if $info \leq n$, the part of $\text{sub}(B)$ containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $\text{sub}(B) = U^H * U$, or $\text{sub}(B) = L * L^H$.

m (global) The total number of eigenvalues found, $0 \leq m \leq n$.

nz (global) Total number of eigenvectors computed. $0 < nz < m$. The number of columns of z that are filled.

If $jobz \neq 'V'$, nz is not referenced.

If $jobz = 'V'$, $nz = m$ unless the user supplies insufficient space and `p?hegvx` is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in z ($m \leq \text{descz}[n_ - 1]$) and sufficient workspace to compute them. (See $lwork$ below.) The function `p?hegvx` is always able to detect insufficient space without computation unless $range = 'V'$.

w (global)

Array of size n . On normal exit, the first m entries contain the selected eigenvalues in ascending order.

z (local).

global size $n * n$, local size $lld_z * LOCC(jz + n - 1)$.

If `jobz = 'V'`, then on normal exit the first m columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in `ifail`.

If `jobz = 'N'`, then z is not referenced.

`work`

On exit, `work[0]` returns the optimal amount of workspace.

`rwork`

On exit, `rwork[0]` contains the amount of workspace required for optimal efficiency

If `jobz='N'` `rwork[0]` = optimal amount of workspace required to compute eigenvalues efficiently

If `jobz='V'` `rwork[0]` = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If `range='V'`, it is assumed that all eigenvectors may be required when computing optimal workspace.

`ifail`

(global)

Array of size n .

`ifail` provides additional information when `info` $\neq 0$

If $(\text{mod}(\text{info}/16, 2) \neq 0)$, then `ifail[0]` indicates the order of the smallest minor which is not positive definite.

If $(\text{mod}(\text{info}, 2) \neq 0)$ on exit, then `ifail[0]` contains the indices of the eigenvectors that failed to converge.

If neither of the above error conditions are held, and `jobz = 'V'`, then the first m elements of `ifail` are set to zero.

`iclustr`

(global)

Array of size $(2 * \text{NPROW} * \text{NPCOL})$. This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see `lwork`, `orfac` and `info`). Eigenvectors corresponding to clusters of eigenvalues indexed `iclustr(2*i-1)` to `iclustr(2*i)`, could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal.

`iclustr()` is a zero terminated array. $(\text{iclustr}(2*k) \neq 0 \text{ and } \text{iclustr}(2*k+1) = 0)$ if and only if k is the number of clusters.

`iclustr` is not referenced if `jobz = 'N'`.

`gap`

(global)

Array of size $\text{NPROW} * \text{NPCOL}$.

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array `iclustr`. As a result, the dot product between eigenvectors corresponding to the i -th cluster may be as high as $(C * n) / \text{gap}(i)$, where C is a small constant.

info

(global)

If *info* = 0, the execution is successful.If *info* < 0: the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.If *info* > 0:If (mod(*info*,2)≠0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.If (mod(*info*,2,2)≠0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.If (mod(*info*/4,2)≠0), then space limit prevented p?sygvx from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.If (mod(*info*/8,2)≠0), then p?stebz failed to compute eigenvalues.If (mod(*info*/16,2)≠0), then *B* was not positive definite. *ifail*(1) indicates the order of the smallest minor which is not positive definite.**See Also**[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.**ScaLAPACK Auxiliary Routines****ScaLAPACK Auxiliary Routines**

Routine Name	Data Types	Description
p?lacgv	c, z	Conjugates a complex vector.
p?max1	c, z	Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS p?amax, but using the absolute value to the real part).
pmpcol	s, d	<i>Finds the collaborators of a process.</i>
pmpim2	s, d	Computes the eigenpair range assignments for all processes.
?combamax1	c, z	Finds the element with maximum real part absolute value and its corresponding global index.
p?sum1	sc, dz	Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum, but using the true absolute value.
p?dbtrsv	s, d, c, z	Computes an <i>LU</i> factorization of a general tridiagonal matrix with no pivoting. The routine is called by p?dbtrs.
p?dttrsv	s, d, c, z	Computes an <i>LU</i> factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by p?dttrs.
p?gebal	s, d	Balances a general real/complex matrix.

Routine Name	Data Types	Description
p?gebd2	s, d, c, z	Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).
p?gehd2	s, d, c, z	Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).
p?gelq2	s, d, c, z	Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).
p?geql2	s, d, c, z	Computes a QL factorization of a general rectangular matrix (unblocked algorithm).
p?geqr2	s, d, c, z	Computes a QR factorization of a general rectangular matrix (unblocked algorithm).
p?gerq2	s, d, c, z	Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).
p?getf2	s, d, c, z	Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).
p?labrd	s, d, c, z	Reduces the first nb rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.
p?lacon	s, d, c, z	Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.
p?laconsb	s, d	Looks for two consecutive small subdiagonal elements.
p?lACP2	s, d, c, z	Copies all or part of a distributed matrix to another distributed matrix.
p?lACP3	s, d	Copies from a global parallel array into a local replicated array or vice versa.
p?lACPy	s, d, c, z	Copies all or part of one two-dimensional array to another.
p?laevswp	s, d, c, z	Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.
p?lahrd	s, d, c, z	Reduces the first nb columns of a general rectangular matrix A so that elements below the k^{th} subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.
p?laiect	s, d, c, z	Exploits IEEE arithmetic to accelerate the computations of eigenvalues.
p?lamve	s, d	Copies all or part of one two-dimensional distributed array to another.
p?lange	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.
p?lanhs	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.

Routine Name	Data Types	Description
p?lansy, p?lanhe	<i>s, d, c, z/c, z</i>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a real symmetric or complex Hermitian matrix.
p?lantr	<i>s, d, c, z</i>	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.
p?lapiv	<i>s, d, c, z</i>	Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.
p?laqge	<i>s, d, c, z</i>	Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ .
p?laqr0	<i>s, d</i>	Computes the eigenvalues of a Hessenberg matrix and optionally returns the matrices from the Schur decomposition.
p?laqr1	<i>s, d</i>	Sets a scalar multiple of the first column of the product of a 2-by-2 or 3-by-3 matrix and specified shifts.
p?laqr2	<i>s, d</i>	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
p?laqr3	<i>s, d</i>	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
p?laqr5	<i>s, d</i>	Performs a single small-bulge multi-shift QR sweep.
p?laqsy	<i>s, d, c, z</i>	Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ .
p?lared1d	<i>s, d</i>	Redistributes an array assuming that the input array <i>bycol</i> is distributed across rows and that all process columns contain the same copy of <i>bycol</i> .
p?lared2d	<i>s, d</i>	Redistributes an array assuming that the input array <i>byrow</i> is distributed across columns and that all process rows contain the same copy of <i>byrow</i> .
p?larf	<i>s, d, c, z</i>	Applies an elementary reflector to a general rectangular matrix.
p?larfb	<i>s, d, c, z</i>	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
p?larfc	<i>c, z</i>	Applies the conjugate transpose of an elementary reflector to a general matrix.
p?larfg	<i>s, d, c, z</i>	Generates an elementary reflector (Householder matrix).
p?larft	<i>s, d, c, z</i>	Forms the triangular vector <i>T</i> of a block reflector $H=I- VTV^H$
p?larz	<i>s, d, c, z</i>	Applies an elementary reflector as returned by p?tzzrf to a general matrix.
p?larzb	<i>s, d, c, z</i>	Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzzrf to a general matrix.

Routine Name	Data Types	Description
p?larzc	<i>c, z</i>	Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzrzf to a general matrix.
p?larzt	<i>s, d, c, z</i>	Forms the triangular factor T of a block reflector $H=I- VTV^H$ as returned by p?tzrzf .
p?lascl	<i>s, d, c, z</i>	Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from} .
p?laset	<i>s, d, c, z</i>	Initializes the off-diagonal elements of a matrix to α and the diagonal elements to β .
p?lasmsub	<i>s, d</i>	Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.
p?lassq	<i>s, d, c, z</i>	Updates a sum of squares represented in scaled form.
p?laswp	<i>s, d, c, z</i>	Performs a series of row interchanges on a general rectangular matrix.
p?latra	<i>s, d, c, z</i>	Computes the trace of a general square distributed matrix.
p?latrd	<i>s, d, c, z</i>	Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.
p?latrz	<i>s, d, c, z</i>	Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.
p?lauu2	<i>s, d, c, z</i>	Computes the product UU^H or L^HL , where U and L are upper or lower triangular matrices (local unblocked algorithm).
p?lauum	<i>s, d, c, z</i>	Computes the product UU^H or L^HL , where U and L are upper or lower triangular matrices.
p?lawil	<i>s, d</i>	Forms the Wilkinson transform.
p?org2l/p?ung2l	<i>s, d, c, z</i>	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by p?geqlf (unblocked algorithm).
p?org2r/p?ung2r	<i>s, d, c, z</i>	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by p?geqrf (unblocked algorithm).
p?orgl2/p?ungl2	<i>s, d, c, z</i>	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf (unblocked algorithm).
p?orgr2/p?ungr2	<i>s, d, c, z</i>	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by p?gerqf (unblocked algorithm).
p?orm2l/p?unm2l	<i>s, d, c, z</i>	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf (unblocked algorithm).
p?orm2r/p?unm2r	<i>s, d, c, z</i>	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by p?geqrf (unblocked algorithm).
p?orml2/p?unml2	<i>s, d, c, z</i>	Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf (unblocked algorithm).
p?ormr2/p?unmr2	<i>s, d, c, z</i>	Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by p?gerqf (unblocked algorithm).

Routine Name	Data Types	Description
p?pbtrsv	s, d, c, z	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by p?pbtrf .
p?pttrsv	s, d, c, z	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf .
p?potf2	s, d, c, z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).
p?rot	s, d	Applies a planar rotation to two distributed vectors.
p?rscl	s, d, cs, zd	Multiplies a vector by the reciprocal of a real scalar.
p?sygs2/p?hegs2	s, d, c, z	Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).
p?syt2/p?hetd2	s, d, c, z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).
p?trord	s, d	Reorders the Schur factorization of a general matrix.
p?trsen	s, d	Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.
p?trti2	s, d, c, z	Computes the inverse of a triangular matrix (local unblocked algorithm).
?lamsh	s, d	Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.
?laqr6	s, d	Performs a single small-bulge multi-shift QR sweep collecting the transformations.
?lar1va	s, d	<i>Computes scaled eigenvector corresponding to given eigenvalue.</i>
?laref	s, d	Applies Householder reflectors to matrices on either their rows or columns.
?larrb2	s, d	Provides limited bisection to locate eigenvalues for more accuracy.
?larnd2	s, d	Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.
?larre2	s, d	Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.
?larre2a	s, d	<i>Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.</i>
?larrf2	s, d	Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.
?larrv2	s, d	Computes the eigenvectors of the tridiagonal matrix $T = L^*D^*L^T$ given L , D and the eigenvalues of $L^*D^*L^T$.

Routine Name	Data Types	Description
?lasorte	s, d	Sorts eigenpairs by real and complex data types.
?lasrt2	s, d	Sorts numbers in increasing or decreasing order.
?stegr2	s, d	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
?stegr2a	s, d	Computes selected eigenvalues and initial representations needed for eigenvector computations.
?stegr2b	s, d	From eigenvalues and initial representations computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors.
?stein2	s, d	Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.
?dbtbf2	s, d, c, z	Computes an <i>LU</i> factorization of a general band matrix with no pivoting (local unblocked algorithm).
?dbtrf	s, d, c, z	Computes an <i>LU</i> factorization of a general band matrix with no pivoting (local blocked algorithm).
?dttrf	s, d, c, z	Computes an <i>LU</i> factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).
?dttrsv	s, d, c, z	Solves a general tridiagonal system of linear equations using the <i>LU</i> factorization computed by ?dttrf .
?pttrsv	s, d, c, z	Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the <i>LDL^H</i> factorization computed by ?pttrf .
?stegr2	s, d	Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit <i>QL</i> or <i>QR</i> method.
?trmvt	s, d, c, z	Performs matrix-vector operations.
pilaenv	NA	Returns the positive integer value of the logical blocking size.
pilaenvx	NA	Called from the ScaLAPACK routines to choose problem-dependent parameters for the local environment.
pjlaenv	NA	Called from the ScaLAPACK symmetric and Hermitian tailored eigen-routines to choose problem-dependent parameters for the local environment.

Product and Performance Information

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

Notice revision #20201201

[p?lacgv](#)

Conjugates a complex vector.

Syntax

```
void p?clacgv (MKL_INT *n , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );

void pzclacgv (MKL_INT *n , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
```

Include Files

- mkl_scalapack.h

Description

The `p?clacgv` function conjugates a complex vector `sub(X)` of length n , where `sub(X)` denotes $X(ix, jx:jx+n-1)$ if `incx = m_x`, and $X(ix:ix+n-1, jx)$ if `incx = 1`.

Input Parameters

n	(global) The length of the distributed vector <code>sub(X)</code> .
x	(local). Pointer into the local memory to an array of size $lld_x * LOCc(n_x)$. On entry the vector to be conjugated $x[i] = X(ix+(jx-1)*m_x+i*incx)$, $0 \leq i < n$.
ix	(global) The row index in the global matrix X indicating the first row of <code>sub(X)</code> .
jx	(global) The column index in the global matrix X indicating the first column of <code>sub(X)</code> .
$descx$	(global and local) Array of size $dlen_ = 9$. The array descriptor for the distributed matrix X .
$incx$	(global) The global increment for the elements of X . Only two values of $incx$ are supported in this version, namely 1 and m_x . $incx$ must not be zero.

Output Parameters

x	(local). On exit, the local pieces of conjugated distributed vector <code>sub(X)</code> .
-----	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?amax1

Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS `p?amax`, but using the absolute value to the real part).

Syntax

```
void p?cmax1 (MKL_INT *n , MKL_Complex8 *amax , MKL_INT *indx , MKL_Complex8 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx );

void pzmax1 (MKL_INT *n , MKL_Complex16 *amax , MKL_INT *indx , MKL_Complex16 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?max1` function computes the global index of the maximum element in absolute value of a distributed vector `sub(X)`. The global index is returned in `indx` and the value is returned in `amax`, where `sub(X)` denotes `X(ix:ix+n-1, jx)` if `incx = 1`, `X(ix, jx:jx+n-1)` if `incx = m_x`.

Input Parameters

<code>n</code>	(global). The number of components of the distributed vector <code>sub(X)</code> . $n \geq 0$.
<code>x</code>	(local) Pointer into the local memory to an array of size <code>lld_x * LOCc(jx+n-1)</code> . On entry this array contains the local pieces of the distributed vector <code>sub(X)</code> .
<code>ix</code>	(global) The row index in the global matrix <code>X</code> indicating the first row of <code>sub(X)</code> .
<code>jx</code>	(global) The column index in the global matrix <code>X</code> indicating the first column of <code>sub(X)</code> .
<code>descx</code>	(global and local) Array of size <code>dlen_</code> . The array descriptor for the distributed matrix <code>X</code> .
<code>incx</code>	(global). The global increment for the elements of <code>X</code> . Only two values of <code>incx</code> are supported in this version, namely 1 and <code>m_x</code> . <code>incx</code> must not be zero.

Output Parameters

<code>amax</code>	(global output). The absolute value of the largest entry of the distributed vector <code>sub(X)</code> only in the scope of <code>sub(X)</code> .
<code>indx</code>	(global output). The global index of the element of the distributed vector <code>sub(X)</code> whose real part has maximum absolute value.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

`pilaver`

Returns the ScaLAPACK version.

Syntax

```
void pilaver (MKL_INT* vers_major, MKL_INT* vers_minor, MKL_INT* vers_patch);
```

Include Files

- `mkl_scalapack.h`

Description

This function returns the ScaLAPACK version.

Output Parameters

<code>vers_major</code>	Return the ScaLAPACK major version.
-------------------------	-------------------------------------

`vers_minor` Return the ScaLAPACK minor version from the major version.

`vers_patch` Return the ScaLAPACK patch version from the minor version.

pmpcol

Finds the collaborators of a process.

Syntax

```
void pmpcol(MKL_INT* myproc, MKL_INT* nprocs, MKL_INT* iil, MKL_INT* needil, MKL_INT*
neediu, MKL_INT* pmyils, MKL_INT* pmyius, MKL_INT* colbrt, MKL_INT* frstcl, MKL_INT*
lastcl);
```

Include Files

- `mkl_scalapack.h`

Description

Using the output from `pmpim2` and given the information on eigenvalue clusters, `pmpcol` finds the collaborators of `myproc`.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

`myproc` The processor number, $0 \leq myproc < nprocs$.

`nprocs` The total number of processors available.

`iil` The index of the leftmost eigenvalue in the eigenvalue cluster.

`needil` The leftmost position in the eigenvalue cluster needed by `myproc`.

`neediu` The rightmost position in the eigenvalue cluster needed by `myproc`.

`pmyils` array
For each processor p , $0 < p \leq nprocs$, `pmyils[p-1]` is the index of the first eigenvalue in the eigenvalue cluster to be computed.
`pmyils[p-1]` equals zero if p stays idle.

`pmyius` array
For each processor p , `pmyius[p-1]` is the index of the last eigenvalue in the eigenvalue cluster to be computed.
`pmyius[p-1]` equals zero if p stays idle.

OUTPUT Parameters

`colbrt` Non-zero if `myproc` collaborates.

`frstcl, lastcl` First and last collaborator of `myproc`.

?combamax1

Finds the element with maximum real part absolute value and its corresponding global index.

Syntax

```
void ccombamax1 (MKL_Complex8 *v1 , MKL_Complex8 *v2 );
void zcombamax1 (MKL_Complex16 *v1 , MKL_Complex16 *v2 );
```

Include Files

- mkl_scalapack.h

Description

The ?combamax1 function finds the element having maximum real part absolute value as well as its corresponding global index.

Input Parameters

<i>v1</i>	(local) Array of size 2. The first maximum absolute value element and its global index. <i>v1</i> [0]=amax, <i>v1</i> [1]=indx.
<i>v2</i>	(local) Array of size 2. The second maximum absolute value element and its global index. <i>v2</i> [0]=amax, <i>v2</i> [1]=indx.

Output Parameters

<i>v1</i>	(local). The first maximum absolute value element and its global index. <i>v1</i> [0]=amax, <i>v1</i> [1]=indx.
-----------	---

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?sum1

Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum, but using the true absolute value.

Syntax

```
void pscsum1 (MKL_INT *n , float *asum , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
void pdzsum1 (MKL_INT *n , double *asum , MKL_Complex16 *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , MKL_INT *incx );
```

Include Files

- mkl_scalapack.h

Description

The p?sum1 function returns the sum of absolute values of a complex distributed vector sub(*x*) in *asum*, where sub(*x*) denotes *X*(*ix:ix+n-1*, *jx:jx*), if *incx* = 1, *X*(*ix:ix*, *jx:jx+n-1*), if *incx* = *m_x*.

Based on `p?asum` from the Level 1 PBLAS. The change is to use the 'genuine' absolute value.

Input Parameters

<code>n</code>	(global). The number of components of the distributed vector <code>sub(x)</code> . $n \geq 0$.
<code>x</code>	(local) Pointer into the local memory to an array of size <code>lld_x * LOCc(j_x+n-1)</code> . This array contains the local pieces of the distributed vector <code>sub(X)</code> .
<code>ix</code>	(global) The row index in the global matrix <code>X</code> indicating the first row of <code>sub(X)</code> .
<code>jx</code>	(global) The column index in the global matrix <code>X</code> indicating the first column of <code>sub(X)</code>
<code>descx</code>	(local) Array of size <code>dlen_=9</code> . The array descriptor for the distributed matrix <code>X</code> .
<code>incx</code>	(global) The global increment for the elements of <code>X</code> . Only two values of <code>incx</code> are supported in this version, namely 1 and <code>m_x</code> .

Output Parameters

<code>asum</code>	(local) The sum of absolute values of the distributed vector <code>sub(X)</code> only in its scope.
-------------------	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dbtrsv

Computes an LU factorization of a general triangular matrix with no pivoting. The function is called by `p?dbtrs`.

Syntax

```
void psdbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu ,
MKL_INT *nrhs , float *a , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib ,
MKL_INT *descb , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT
*info );

void pddbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu ,
MKL_INT *nrhs , double *a , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *descb , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT
*info );

void pcdbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu ,
MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzdbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu ,
MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?dbtrsv` function solves a banded triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs) \text{ or}$$

$$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs) \text{ (for real flavors); } A(1:n, ja:ja+n-1)^H * X = B(ib:ib+n-1, 1:nrhs) \text{ (for complex flavors),}$$

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Gaussian elimination code of `p?dbtrf` and is stored in $A(1:n, ja:ja+n-1)$ and af . The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to $uplo$, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter $trans$.

The function `p?dbtrf` must be called first.

Input Parameters

<code>uplo</code>	(global) If <code>uplo='U'</code> , the upper triangle of $A(1:n, ja:ja+n-1)$ is stored, if <code>uplo='L'</code> , the lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<code>trans</code>	(global) If <code>trans='N'</code> , solve with $A(1:n, ja:ja+n-1)$, if <code>trans='C'</code> , solve with conjugate transpose $A(1:n, ja:ja+n-1)$.
<code>n</code>	(global) The order of the distributed submatrix A ; ($n \geq 0$).
<code>bwl</code>	(global) Number of subdiagonals. $0 \leq bwl \leq n-1$.
<code>bwu</code>	(global) Number of subdiagonals. $0 \leq bwu \leq n-1$.
<code>nrhs</code>	(global) The number of right-hand sides; the number of columns of the distributed submatrix B ($nrhs \geq 0$).
<code>a</code>	(local). Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$, where $lld_a \geq (bwl+bwu+1)$. On entry, this array contains the local pieces of the n -by- n unsymmetric banded distributed Cholesky factor L or L^T , represented in global A as $A(1:n, ja:ja+n-1)$. This local portion is stored in the packed banded format used in LAPACK. See the <i>Application Notes</i> below and the ScaLAPACK manual for more detail on the format of distributed matrices.
<code>ja</code>	(global) The index in the global matrix A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<code>desca</code>	(global and local) array of size $dlen_$. if 1d type ($dtype_a = 501$ or 502), $dlen \geq 7$; if 2d type ($dtype_a = 1$), $dlen \geq 9$. The array descriptor for the distributed matrix A . Contains information of mapping of A to memory.

<i>b</i>	(local) Pointer into the local memory to an array of local lead dimension $lld_b \geq nb$. On entry, this array contains the local pieces of the right-hand sides $B(ib:ib+n-1, 1:nrhs)$.
<i>ib</i>	(global) The row index in the global matrix B that points to the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).
<i>descb</i>	(global and local) array of size $dlen_$. if 1d type ($dtype_b = 502$), $dlen \geq 7$; if 2d type ($dtype_b = 1$), $dlen \geq 9$. The array descriptor for the distributed matrix B . Contains information of mapping B to memory.
<i>laf</i>	(local) Size of user-input auxiliary fill-in space af . $laf \geq nb * (bwl + bwu) + 6 * \max(bwl, bwu) * \max(bwl, bwu)$. If laf is not large enough, an error code is returned and the minimum acceptable size will be returned in $af[0]$.
<i>work</i>	(local). Temporary workspace. This space may be overwritten in between function calls. <i>work</i> must be the size given in <i>lwork</i> .
<i>lwork</i>	(local or global) Size of user-input workspace <i>work</i> . If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work[0]</i> and an error code is returned. $lwork \geq \max(bwl, bwu) * nrhs$.

Output Parameters

<i>a</i>	(local). This local portion is stored in the packed banded format used in LAPACK. Please see the ScaLAPACK manual for more detail on the format of distributed matrices.
<i>b</i>	On exit, this contains the local piece of the solutions distributed matrix X .
<i>af</i>	(local). auxiliary fill-in space. The fill-in space is created in a call to the factorization function <code>p?dbtrf</code> and is stored in <i>af</i> . If a linear system is to be solved using <code>p?dbtrf</code> after the factorization function, <i>af</i> must not be altered after the factorization.
<i>work</i>	On exit, <i>work[0]</i> contains the minimal <i>lwork</i> .
<i>info</i>	(local). If <i>info</i> = 0, the execution is successful.

< 0: If the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?dttrs

Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges. The function is called by p?dttrs.

Syntax

```
void psdttrs (char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, float *dl, float *d, float *du, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *descb, float *af, MKL_INT *laf, float *work, MKL_INT *lwork, MKL_INT *info);

void pddttrs (char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, double *dl, double *d, double *du, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *descb, double *af, MKL_INT *laf, double *work, MKL_INT *lwork, MKL_INT *info);

void pcdttrs (char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, MKL_Complex8 *dl, MKL_Complex8 *d, MKL_Complex8 *du, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *af, MKL_INT *laf, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);

void pzdttrs (char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, MKL_Complex16 *dl, MKL_Complex16 *d, MKL_Complex16 *du, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

Include Files

- mkl_scalapack.h

Description

The p?dttrs function solves a tridiagonal triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs) \text{ or}$$

$$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs) \text{ for real flavors; } A(1:n, ja:ja+n-1)^H * X = B(ib:ib+n-1, 1:nrhs) \text{ for complex flavors,}$$

where $A(1:n, ja:ja+n-1)$ is a tridiagonal matrix factor produced by the Gaussian elimination code of p?dttrf and is stored in $A(1:n, ja:ja+n-1)$ and af .

The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to $uplo$, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter $trans$.

The function p?dttrf must be called first.

Input Parameters

$uplo$ (global)
If $uplo = 'U'$, the upper triangle of $A(1:n, ja:ja+n-1)$ is stored,

	if <i>uplo</i> = 'L', the lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<i>trans</i>	(global) If <i>trans</i> = 'N', solve with $A(1:n, ja:ja+n-1)$, if <i>trans</i> = 'C', solve with conjugate transpose $A(1:n, ja:ja+n-1)$.
<i>n</i>	(global) The order of the distributed submatrix <i>A</i> ; ($n \geq 0$).
<i>nrhs</i>	(global) The number of right-hand sides; the number of columns of the distributed submatrix $B(ib:ib+n-1, 1:nrhs)$. ($nrhs \geq 0$).
<i>dl</i>	(local). Pointer to local part of global vector storing the lower diagonal of the matrix. Globally, <i>dl</i> [0] is not referenced, and <i>dl</i> must be aligned with <i>d</i> . Must be of size $\geq nb_a$.
<i>d</i>	(local). Pointer to local part of global vector storing the main diagonal of the matrix.
<i>du</i>	(local). Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, <i>du</i> [<i>n</i> -1] is not referenced, and <i>du</i> must be aligned with <i>d</i> .
<i>ja</i>	(global) The index in the global matrix <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) array of size <i>dlen_</i> . if 1 <i>d</i> type (<i>dtype_a</i> = 501 or 502), <i>dlen</i> ≥ 7 ; if 2 <i>d</i> type (<i>dtype_a</i> = 1), <i>dlen</i> ≥ 9 . The array descriptor for the distributed matrix <i>A</i> . Contains information of mapping of <i>A</i> to memory.
<i>b</i>	(local) Pointer into the local memory to an array of local lead dimension $lld_b \geq nb$. On entry, this array contains the local pieces of the right-hand sides $B(ib:ib+n-1, 1:nrhs)$.
<i>ib</i>	(global) The row index in the global matrix <i>B</i> that points to the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) array of size <i>dlen_</i> . if 1 <i>d</i> type (<i>dtype_b</i> = 502), <i>dlen</i> ≥ 7 ; if 2 <i>d</i> type (<i>dtype_b</i> = 1), <i>dlen</i> ≥ 9 . The array descriptor for the distributed matrix <i>B</i> . Contains information of mapping <i>B</i> to memory.
<i>laf</i>	(local).

Size of user-input auxiliary fill-in space *af*.

$laf \geq 2 * (nb + 2)$. If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*[0].

work

(local).

Temporary workspace. This space may be overwritten in between function calls.

work must be the size given in *lwork*.

lwork

(local or global)

Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*[0] and an error code is returned.

$lwork \geq 10 * npcol + 4 * nrhs$.

Output Parameters

dl

(local).

On exit, this array contains information containing the factors of the matrix.

d

On exit, this array contains information containing the factors of the matrix. Must be of size $\geq nb_a$.

b

On exit, this contains the local piece of the solutions distributed matrix X.

af

(local).

Auxiliary fill-in space. The fill-in space is created in a call to the factorization function `p?dtttrf` and is stored in *af*. If a linear system is to be solved using `p?dtttrs` after the factorization function, *af* must not be altered after the factorization.

work

On exit, *work*[0] contains the minimal *lwork*.

info

(local).

If *info*=0, the execution is successful.

if *info*< 0: If the *i*-th argument is an array and the *j*-th entry, indexed *j*-1, had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gebal

Balances a general real/complex matrix.

Syntax

```
void psgebal(char* job, MKL_INT* n, float* a, MKL_INT* desca, MKL_INT* ilo, MKL_INT*
ihi, float* scale, MKL_INT* info);
```

```
void pdgebal(char* job, MKL_INT* n, double* a, MKL_INT* desca, MKL_INT* ilo, MKL_INT*
ihi, double* scale, MKL_INT* info);
```

```
void pcgebal(char* job, MKL_INT* n, complex float* a, MKL_INT* desca, MKL_INT* ilo,
MKL_INT* ihi, float* scale, MKL_INT* info);
```

```
void pzgebal(char* job, MKL_INT* n, complex double* a, MKL_INT* desca, MKL_INT* ilo,
MKL_INT* ihi, double* scale, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

pzgebal balances a general real/complex matrix A . This involves, first, permuting A by a similarity transformation to isolate eigenvalues in the first 1 to $ilo-1$ and last $ihi+1$ to n elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ilo to ihi to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrix, and improve the accuracy of the computed eigenvalues and/or eigenvectors.

Input Parameters

<i>job</i>	(global) Specifies the operations to be performed on a : = 'N': none: simply set $ilo = 1$, $ihi = n$, $scale[i] = 1.0$ for $i = 0, \dots, n-1$; = 'P': permute only; = 'S': scale only; = 'B': both permute and scale.
<i>n</i>	(global) The order of the matrix A ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOC_c(n)$ This array contains the local pieces of global input matrix A .
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

OUTPUT Parameters

<i>a</i>	On exit, a is overwritten by the balanced matrix A . If $job = 'N'$, a is not referenced. See Notes for further details.
<i>ilo, ihi</i>	(global) ilo and ihi are set to integers such that on exit matrix elements $A(i,j)$ are zero if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$. If $job = 'N'$ or 'S', $ilo = 1$ and $ihi = n$.
<i>scale</i>	(global) array of size n . Details of the permutations and scaling factors applied to a . If pj is the index of the row and column interchanged with row and column j and dj is the scaling factor applied to row and column j , then $scale[j-1] = pj$ for $j = 1, \dots, ilo-1, ihi+1, \dots, n$

$scale[j-1] = dj$ for $j = ilo, \dots, ihi$

The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

info

(global)

= 0: successful exit.

< 0: if $info = -i$, the i -th argument had an illegal value.

Application Notes

The permutations consist of row and column interchanges which put the matrix in the form

$$PAP = \begin{pmatrix} T_1 & X & Y \\ 0 & B & Z \\ 0 & 0 & T_2 \end{pmatrix}$$

where T_1 and T_2 are upper triangular matrices whose eigenvalues lie along the diagonal. The column indices ilo and ihi mark the starting and ending columns of the submatrix B . Balancing consists of applying a diagonal similarity transformation $D^{-1}BD$ to make the 1-norms of each row of B and its corresponding column nearly equal. The output matrix is

$$\begin{pmatrix} T_1 & XD & Y \\ 0 & D^{-1}BD & D^{-1}Z \\ 0 & 0 & T_2 \end{pmatrix}$$

Information about the permutations P and the diagonal matrix D is returned in the vector *scale*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gebd2

Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).

Syntax

```
void psgebd2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *d , float *e , float *tauq , float *taup , float *work , MKL_INT
*lwork , MKL_INT *info );

void pdgebd2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *d , double *e , double *tauq , double *taup , double *work , MKL_INT
*lwork , MKL_INT *info );

void pcgebd2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *d , float *e , MKL_Complex8 *tauq , MKL_Complex8 *taup ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzgebd2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *d , double *e , MKL_Complex16 *tauq , MKL_Complex16 *taup ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?gebd2 function reduces a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form B by an orthogonal/unitary transformation:

$$Q' * \text{sub}(A) * P = B.$$

If $m \geq n$, B is the upper bidiagonal; if $m < n$, B is the lower bidiagonal.

Input Parameters

<i>m</i>	(global) The number of rows of the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). Pointer into the local memory to an array of size $\text{lld}_a * \text{LOC}_c(ja+n-1)$. On entry, this array contains the local pieces of the general distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>work</i>	(local). This is a workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $\text{lwork} \geq \max(\text{mpa0}, \text{nqa0})$, where $\text{nb} = \text{mb}_a = \text{nb}_a$, $\text{iroffa} = \text{mod}(\text{ia}-1, \text{nb})$, $\text{iarow} = \text{indxg2p}(\text{ia}, \text{nb}, \text{myrow}, \text{rsrc}_a, \text{nprow})$, $\text{iacol} = \text{indxg2p}(\text{ja}, \text{nb}, \text{mycol}, \text{csrc}_a, \text{npcol})$, $\text{mpa0} = \text{numroc}(\text{m} + \text{iroffa}, \text{nb}, \text{myrow}, \text{iarow}, \text{nprow})$, $\text{nqa0} = \text{numroc}(\text{n} + \text{icoffa}, \text{nb}, \text{mycol}, \text{iacol}, \text{npcol})$. indxg2p and numroc are ScaLAPACK tool functions; <i>myrow</i> , <i>mycol</i> , <i>nprow</i> , and <i>npcol</i> can be determined by calling the function <code>blacs_gridinfo</code> . If $\text{lwork} = -1$, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pxerbla</code> .

Output Parameters

<i>a</i>	(local). On exit, if $m \geq n$, the diagonal and the first superdiagonal of $\text{sub}(A)$ are overwritten with the upper bidiagonal matrix B ; the elements below the diagonal, with the array <i>tauq</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array <i>taup</i> , represent the orthogonal matrix P as a product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B ; the elements below the first subdiagonal, with the array <i>tauq</i> , represent the
----------	--

orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array $taup$, represent the orthogonal matrix P as a product of elementary reflectors. See *Applications Notes* below.

d	(local) Array of size $LOCc(ja+\min(m,n)-1)$ if $m \geq n$; $LOCr(ia+\min(m,n)-1)$ otherwise. The distributed diagonal elements of the bidiagonal matrix B : $d[i] = A(i+1,i+1)$, $i=0, 1, \dots$, size $(d) - 1$. d is tied to the distributed matrix A .
e	(local) Array of size $LOCc(ja+\min(m,n)-1)$ if $m \geq n$; $LOCr(ia+\min(m,n)-2)$ otherwise. The distributed diagonal elements of the bidiagonal matrix B : if $m \geq n$, $e[i] = A(i+1,i+2)$ for $i = 0, 1, \dots, n-2$; if $m < n$, $e[i] = A(i+2,i+1)$ for $i = 0, 1, \dots, m-2$. e is tied to the distributed matrix A .
$tauq$	(local). Array of size $LOCc(ja+\min(m,n)-1)$. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q . $tauq$ is tied to the distributed matrix A .
$taup$	(local). Array of size $LOCr(ia+\min(m,n)-1)$. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix P . $taup$ is tied to the distributed matrix A .
$work$	On exit, $work[0]$ returns the minimal and optimal $lwork$.
$info$	(local) If $info = 0$, the execution is successful. if $info < 0$: If the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

If $m \geq n$,

$$Q = H(1) * H(2) * \dots * H(n), \text{ and } P = G(1) * G(2) * \dots * G(n-1)$$

Each $H(i)$ and $G(i)$ has the form:

$$H(i) = I - tauq * v * v', \text{ and } G(i) = I - taup * u * u',$$

where $tauq$ and $taup$ are real/complex scalars, and v and u are real/complex vectors. $v(1:i-1) = 0$, $v(i) = 1$, and $v(i+1:m)$ is stored on exit in

$$A(ia+i-ia+m-1, ja+i-1);$$

$$u(1:i) = 0, u(i+1) = 1, \text{ and } u(i+2:n) \text{ is stored on exit in } A(ia+i-1, ja+i+1:ja+n-1);$$

τ_{uq} is stored in $\tau_{uq}[ja+i-2]$ and τ_{up} in $\tau_{up}[ia+i-2]$.

If $m < n$,

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(ia+i+1:ia+m-1, ja+i-1)$;

$u(1:i-1) = 0$, $u(i) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$;

τ_{uq} is stored in $\tau_{uq}[ja+i-2]$ and τ_{up} in $\tau_{up}[ia+i-2]$.

The contents of sub(A) on exit are illustrated by the following examples:

$m = 6$ and $n = 5 (m > n)$:

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5$ and $n = 6 (m < n)$:

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of B , v_i denotes an element of the vector defining $H(i)$, and u_i an element of the vector defining $G(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gehd2

Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).

Syntax

```
void psgehd2 (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT
*info );
```

```
void pdgehd2 (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT
*info );
```

```
void pcgehd2 (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
```

```
void pzgehd2 (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?gehd2` function reduces a real/complex general distributed matrix $\text{sub}(A)$ to upper Hessenberg form H by an orthogonal/unitary similarity transformation: $Q^* \text{sub}(A) Q = H$, where $\text{sub}(A) = A(\text{ia}+n-1 : \text{ia}+n-1, \text{ja}+n-1 : \text{ja}+n-1)$.

Input Parameters

<code>n</code>	(global) The order of the distributed submatrix A . ($n \geq 0$).
<code>ilo, ihi</code>	(global) It is assumed that the matrix $\text{sub}(A)$ is already upper triangular in rows $\text{ia}:\text{ia}+\text{ilo}-2$ and $\text{ia}+\text{ihi}:\text{ia}+n-1$ and columns $\text{ja}:\text{ja}+\text{jlo}-2$ and $\text{ja}+\text{jhi}:\text{ja}+n-1$. See <i>Application Notes</i> for further information. If $n \geq 0, 1 \leq \text{ilo} \leq \text{ihi} \leq n$; otherwise set $\text{ilo} = 1, \text{ihi} = n$.
<code>a</code>	(local). Pointer into the local memory to an array of size $\text{size_lld_a} * \text{LOC}_c(\text{ja}+n-1)$. On entry, this array contains the local pieces of the n -by- n general distributed matrix $\text{sub}(A)$ to be reduced.
<code>ia, ja</code>	(global) The row and column indices in the global matrix A indicating the first row and the first column of $\text{sub}(A)$, respectively.
<code>desca</code>	(global and local) array of size dlen_ . The array descriptor for the distributed matrix A .
<code>work</code>	(local). This is a workspace array of size lwork .
<code>lwork</code>	(local or global) The size of the array work . lwork is local input and must be at least $\text{lwork} \geq \text{nb} + \max(\text{npa0}, \text{nb})$, where $\text{nb} = \text{mb_a} = \text{nb_a}$, $\text{iroffa} = \text{mod}(\text{ia}-1, \text{nb})$, $\text{iarow} = \text{indxg2p}(\text{ia}, \text{nb}, \text{myrow}, \text{rsrc_a}, \text{nprow})$, $\text{npa0} = \text{numroc}(\text{ihi} + \text{iroffa}, \text{nb}, \text{myrow}, \text{iarow}, \text{nprow})$. indxg2p and numroc are ScaLAPACK tool functions; myrow , mycol , nprow , and npcol can be determined by calling the function <code>blacs_gridinfo</code> . If $\text{lwork} = -1$, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>p?xerbla</code> .

Output Parameters

<code>a</code>	(local). On exit, the upper triangle and the first subdiagonal of $\text{sub}(A)$ are overwritten with the upper Hessenberg matrix H , and the elements below the first subdiagonal, with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors. (see <i>Application Notes</i> below).
<code>tau</code>	(local).

Array of size $LOCc(ja+n-2)$ The scalar factors of the elementary reflectors (see *Application Notes* below). Elements $ja:ja+ilo-2$ and $ja+ihi:ja+n-2$ of the global vector τ are set to zero. τ is tied to the distributed matrix A .

work

On exit, *work*[0] returns the minimal and optimal *lwork*.

info

(local)

If *info* = 0, the execution is successful.

if *info* < 0: If the *i*-th argument is an array and the *j*-th entry, indexed *j*-1, had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix Q is represented as a product of $(ihi-ilo)$ elementary reflectors

$$Q = H(ilo) * H(ilo+1) * \dots * H(ihi-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v',$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i)=0$, $v(i+1)=1$ and $v(ihi+1:n)=0$; $v(i+2:ihi)$ is stored on exit in $A(ia+ilo+i:ia+ihi-1, ia+ilo+i-2)$, and τ in $\tau[ja+ilo+i-3]$.

The contents of $A(ia:ia+n-1, ja:ja+n-1)$ are illustrated by the following example, with $n = 7$, $ilo = 2$ and $ihi = 6$:

on entry	on exit
$\begin{bmatrix} a & a & a & a & a & a & a \\ & a & a & a & a & a & a \\ & & a & a & a & a & a \\ & & & a & a & a & a \\ & & & & a & a & a \\ & & & & & a & a \\ & & & & & & a \end{bmatrix}$	$\begin{bmatrix} a & a & h & h & h & h & a \\ & a & h & h & h & h & a \\ & & h & h & h & h & h \\ & & v2 & h & h & h & h \\ & & v2 & v3 & h & h & h \\ & & v2 & v3 & v4 & h & h \\ & & & & & & a \end{bmatrix}$

where a denotes an element of the original matrix $\text{sub}(A)$, h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(ja+ilo+i-2)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gelq2

Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
void psgelq2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdgelq2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

void pcgelq2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzgelq2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- mkl_scalapack.h

Description

The `p?gelq2` function computes an LQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = L*Q$.

Input Parameters

<i>m</i>	(global) The number of rows of the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) The number of columns of the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$. On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ which is to be factored.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>work</i>	(local). This is a workspace array of size $lwork$.
<i>lwork</i>	(local or global) The size of the array <i>work</i> . $lwork$ is local input and must be at least $lwork \geq nq0 + \max(1, mp0)$, where $irow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprw)$, $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcw)$, $mp0 = \text{numroc}(m+irow, mb_a, myrow, irow, nprw)$,

`nq0 = numroc(n+icoff, nb_a, mycol, iacol, npc0l),`
`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<code>a</code>	(local). On exit, the elements on and below the diagonal of <code>sub(A)</code> contain the m by $\min(m,n)$ lower trapezoidal matrix L (L is lower triangular if $m \leq n$); the elements above the diagonal, with the array <code>tau</code> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<code>tau</code>	(local). Array of size <code>LOCr(ia+min(m, n)-1)</code> . This array contains the scalar factors of the elementary reflectors. <code>tau</code> is tied to the distributed matrix A .
<code>work</code>	On exit, <code>work[0]</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local) If <code>info = 0</code> , the execution is successful. if <code>info < 0</code> : If the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then <code>info = -(i*100+j)</code> , if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ia+k-1) * H(ia+k-2) * \dots * H(ia)$ for real flavors, $Q = (H(ia+k-1))^{H*} (H(ia+k-2))^{H*} \dots (H(ia))^{H*}$ for complex flavors,

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - \tau v v'$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ (for real flavors) or $\text{conjg}(v(i+1:n))$ (for complex flavors) is stored on exit in `A(ia+i-1, ja+i:ja+n-1)`, and τ in `tau[ia+i-2]`.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?geql2

Computes a QL factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
void psgeql2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
```



```

void pdgeql2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

void pcgeql2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzgeql2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );

```

Include Files

- mkl_scalapack.h

Description

The `p?geql2` function computes a *QL* factorization of a real/complex distributed *m*-by-*n* matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q * L$.

Input Parameters

<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). Pointer into the local memory to an array of size $lld_a * LOC_c(ja+n-1)$. On entry, this array contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix $\text{sub}(A)$ which is to be factored.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local). This is a workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq mp0 + \max(1, nq0)$, where $iroff = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$, $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npc0l)$, $mp0 = \text{numroc}(m+iroff, mb_a, myrow, iarow, nprow)$, $nq0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npc0l)$, <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; <i>myrow</i> , <i>mycol</i> , <i>nprow</i> , and <i>npc0l</i> can be determined by calling the function <code>blacs_gridinfo</code> .

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a	(local). On exit, if $m \geq n$, the lower triangle of the distributed submatrix $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the n -by- n lower triangular matrix L ; if $m \leq n$, the elements on and below the $(n-m)$ -th superdiagonal contain the m -by- n lower trapezoidal matrix L ; the remaining elements, with the array τ , represent the orthogonal/ unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
τ	(local). Array of size $LOCc(ja+n-1)$. This array contains the scalar factors of the elementary reflectors. τ is tied to the distributed matrix A .
$work$	On exit, $work[0]$ returns the minimal and optimal $lwork$.
$info$	(local). If $info = 0$, the execution is successful. if $info < 0$: If the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja+k-1) * \dots * H(ja+1) * H(ja), \text{ where } k = \min(m, n).$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(m-k+i+1:m) = 0$ and $v(m-k+i) = 1$; $v(1:m-k+i-1)$ is stored on exit in $A(ia:ia+m-k+i-2, ja+n-k+i-1)$, and τ in $\tau[ja+n-k+i-2]$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?geqr2

Computes a QR factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
void psgeqr2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdgeqr2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

```
void pcgeqr2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzgeqr2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- mkl_scalapack.h

Description

The `p?geqr2` function computes a *QR* factorization of a real/complex distributed *m*-by-*n* matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q \cdot R$.

Input Parameters

<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). Pointer into the local memory to an array of size $lld_a * LOC_c(ja+n-1)$. On entry, this array contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix $\text{sub}(A)$ which is to be factored.
<i>ia, ja</i>	(global) The row and column indices in the global matrix <i>A</i> indicating the first row and the first column of $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local). This is a workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq mp0 + \max(1, nq0)$, where $iroff = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$, $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npc0l)$, $mp0 = \text{numroc}(m+iroff, mb_a, myrow, iarow, nprow)$, $nq0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npc0l)$. <code>indxg2p</code> and <code>numroc</code> are ScaLAPACK tool functions; <i>myrow</i> , <i>mycol</i> , <i>nprow</i> , and <i>npcol</i> can be determined by calling the function <code>blacs_gridinfo</code> .

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<code>a</code>	(local). On exit, the elements on and above the diagonal of sub(<i>A</i>) contain the $\min(m,n)$ by n upper trapezoidal matrix <i>R</i> (<i>R</i> is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors (see <i>Application Notes</i> below).
<code>tau</code>	(local). Array of size <code>LOCc(ja+min(m,n)-1)</code> . This array contains the scalar factors of the elementary reflectors. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<code>work</code>	On exit, <code>work[0]</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local) If <code>info = 0</code> , the execution is successful. if <code>info < 0</code> : If the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> -1, had an illegal value, then <code>info = - (i*100+j)</code> , if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

$Q = H(ja) * H(ja+1) * . . . * H(ja+k-1)$, where $k = \min(m,n)$.

Each $H(i)$ has the form

$H(j) = I - \tau * v * v'$,

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$, and *tau* in $\tau[ja+i-2]$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?gerq2

Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```
void psgerq2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdgerq2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

void pcgerq2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );
```

```
void pzgerq2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- mkl_scalapack.h

Description

The `pzgerq2` function computes an RQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = R*Q$.

Input Parameters

<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). Pointer into the local memory to an array of size $lld_a * LOC_c(ja+n-1)$. On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ which is to be factored.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>work</i>	(local). This is a workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq nq0 + \max(1, mp0)$, where $iroff = \text{mod}(ia-1, mb_a), \quad icoff = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow),$ $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol), \quad mp0 = \text{numroc}(m+iroff, mb_a, myrow, iarow, nprow),$ $nq0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npcol),$ $\text{indxg2p} \text{ and } \text{numroc} \text{ are ScaLAPACK tool functions; } myrow, mycol, nprow, \text{ and } npcol \text{ can be determined by calling the function } \text{blacs_gridinfo}.$ If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pxerbla</code> .

Output Parameters

<i>a</i>	(local).
----------	----------

On exit,

if $m \leq n$, the upper triangle of $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the m -by- m upper triangular matrix R ;

if $m \geq n$, the elements on and above the $(m-n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array τ , represent the orthogonal/ unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

τ

(local).

Array of size $LOCr(ia+m-1)$. This array contains the scalar factors of the elementary reflectors. τ is tied to the distributed matrix A .

$work$

On exit, $work[0]$ returns the minimal and optimal $lwork$.

$info$

(local)

If $info = 0$, the execution is successful.

if $info < 0$: If the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ia) * H(ia+1) * \dots * H(ia+k-1)$ for real flavors,

$Q = (H(ia))^H * (H(ia+1))^H * \dots * (H(ia+k-1))^H$ for complex flavors,

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - \tau * v * v'$,

where τ is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ for real flavors or $\text{conjg}(v(1:n-k+i-1))$ for complex flavors is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and τ in $\tau[ia+m-k+i-2]$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?getf2

Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).

Syntax

```
void psgetf2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , MKL_INT *info );

void pdgetf2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , MKL_INT *info );

void pcgetf2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
```

```
void pzgetf2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `pzgetf2` function computes an LU factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ using partial pivoting with row interchanges.

The factorization has the form $\text{sub}(A) = P * L * U$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and U is upper triangular (upper trapezoidal if $m < n$). This is the right-looking Parallel Level 2 BLAS version of the algorithm.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

m	(global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
n	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($nb_a - \text{mod}(ja-1, nb_a) \geq n \geq 0$).
a	(local). Pointer into the local memory to an array of size $lld_a * LOC_c(ja+n-1)$. On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

$ipiv$	(local) Array of size $(LOC_r(m_a) + mb_a)$. This array contains the pivoting information. $ipiv[i] \rightarrow$ The global row that local row $(i+1)$ was swapped with, $i = 0, 1, \dots, LOC_r(m_a) + mb_a - 1$. This array is tied to the distributed matrix A .
$info$	(local). If $info = 0$: successful exit. If $info < 0$:

- if the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$,
- if the i -th argument is a scalar and had an illegal value, then $info = -i$.

If $info > 0$: If $info = k$, the matrix element $U(ia+k-1, ja+k-1)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?labrd

Reduces the first nb rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A .

Syntax

```
void pslabrd (MKL_INT *m , MKL_INT *n , MKL_INT *nb , float *a , MKL_INT *ia , MKL_INT
 *ja , MKL_INT *desca , float *d , float *e , float *tauq , float *taup , float *x ,
 MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *y , MKL_INT *iy , MKL_INT *jy ,
 MKL_INT *descy , float *work );

void pdlabrd (MKL_INT *m , MKL_INT *n , MKL_INT *nb , double *a , MKL_INT *ia , MKL_INT
 *ja , MKL_INT *desca , double *d , double *e , double *tauq , double *taup , double *x ,
 MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *y , MKL_INT *iy , MKL_INT *jy ,
 MKL_INT *descy , double *work );

void pclabrd (MKL_INT *m , MKL_INT *n , MKL_INT *nb , MKL_Complex8 *a , MKL_INT *ia ,
 MKL_INT *ja , MKL_INT *desca , float *d , float *e , MKL_Complex8 *tauq , MKL_Complex8
 *taup , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_Complex8 *y ,
 MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_Complex8 *work );

void pzlabrd (MKL_INT *m , MKL_INT *n , MKL_INT *nb , MKL_Complex16 *a , MKL_INT *ia ,
 MKL_INT *ja , MKL_INT *desca , double *d , double *e , MKL_Complex16 *tauq ,
 MKL_Complex16 *taup , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
 MKL_Complex16 *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description

The `p?labrd` function reduces the first nb rows and columns of a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form by an orthogonal/unitary transformation $Q^* A P$, and returns the matrices X and Y necessary to apply the transformation to the unreduced part of $\text{sub}(A)$.

If $m \geq n$, $\text{sub}(A)$ is reduced to upper bidiagonal form; if $m < n$, $\text{sub}(A)$ is reduced to lower bidiagonal form.

This is an auxiliary function called by `p?gebrd`.

Input Parameters

<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
<i>nb</i>	(global) The number of leading rows and columns of $\text{sub}(A)$ to be reduced.
<i>a</i>	(local). Pointer into the local memory to an array of size $lld_a * LOC_c(ja+n-1)$. On entry, this array contains the local pieces of the general distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>ix, jx</i>	(global) The row and column indices in the global matrix X indicating the first row and the first column of the matrix $\text{sub}(X)$, respectively.
<i>descx</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix X .
<i>iy, jy</i>	(global) The row and column indices in the global matrix Y indicating the first row and the first column of the matrix $\text{sub}(Y)$, respectively.
<i>descy</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix Y .
<i>work</i>	(local). Workspace array of size $lwork$. $lwork \geq nb_a + nq$, with $nq = \text{numroc}(n + \text{mod}(ia-1, nb_y), nb_y, mycol, iacol, npcol)$ $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol)$ indxg2p and numroc are ScaLAPACK tool functions; $myrow$, $mycol$, $nprow$, and $npcol$ can be determined by calling the function <code>blacs_gridinfo</code> .

Output Parameters

<i>a</i>	(local) On exit, the first nb rows and columns of the matrix are overwritten; the rest of the distributed matrix $\text{sub}(A)$ is unchanged. If $m \geq n$, elements on and below the diagonal in the first nb columns, with the array tauq , represent the orthogonal/unitary matrix Q as a product of elementary reflectors; and elements above the diagonal in the first nb rows, with the array taup , represent the orthogonal/unitary matrix P as a product of elementary reflectors.
----------	--

If $m < n$, elements below the diagonal in the first nb columns, with the array *tauq*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements on and above the diagonal in the first nb rows, with the array *taup*, represent the orthogonal/unitary matrix P as a product of elementary reflectors. See *Application Notes* below.

d

(local).

Array of size $LOCr(ia+\min(m,n)-1)$ if $m \geq n$; $LOCc(ja+\min(m,n)-1)$ otherwise. The distributed diagonal elements of the bidiagonal distributed matrix B :

$$d[i] = A(ia+i, ja+i), i = 0, 1, \dots, \text{size}(d)-1$$

d is tied to the distributed matrix A .

e

(local).

Array of size $LOCr(ia+\min(m,n)-1)$ if $m \geq n$; $LOCc(ja+\min(m,n)-2)$ otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix B :

$$\text{if } m \geq n, e[i] = A(ia+i, ja+i+1) \text{ for } i = 0, 1, \dots, n-2;$$

$$\text{if } m < n, e[i] = A(ia+i+1, ja+i) \text{ for } i = 0, 1, \dots, m-2.$$

e is tied to the distributed matrix A .

tauq, taup

(local).

Array size $LOCc(ja+\min(m,n)-1)$ for *tauq*, size $LOCr(ia+\min(m,n)-1)$ for *taup*. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q for *tauq*, P for *taup*. *tauq* and *taup* are tied to the distributed matrix A . See *Application Notes* below.

x

(local)

Pointer into the local memory to an array of size $lld_x * nb$. On exit, the local pieces of the distributed m -by- nb matrix $X(ix:ix+m-1, jx:jx+nb-1)$ required to update the unreduced part of $\text{sub}(A)$.

y

(local).

Pointer into the local memory to an array of size $lld_y * nb$. On exit, the local pieces of the distributed n -by- nb matrix $Y(iy:iy+n-1, jy:jy+nb-1)$ required to update the unreduced part of $\text{sub}(A)$.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

$$Q = H(1) * H(2) * \dots * H(nb), \text{ and } P = G(1) * G(2) * \dots * G(nb)$$

Each $H(i)$ and $G(i)$ has the form:

$$H(i) = I - \text{tauq} * v * v', \text{ and } G(i) = I - \text{taup} * u * u',$$

where *tauq* and *taup* are real/complex scalars, and v and u are real/complex vectors.

If $m \geq n$, $v(1:i-1) = 0$, $v(i) = 1$, and $v(i:m)$ is stored on exit in

$A(ia+i-1:ia+m-1, ja+i-1)$; $u(1:i) = 0$, $u(i+1) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$; *tauq* is stored in *tauq*[*ja+i-2*] and *taup* in *taup*[*ia+i-2*].

If $m < n$, $v(1:i) = 0$, $v(i+1) = 1$, and $v(i+1:m)$ is stored on exit in

$A(ia+i+1:ia+m-1, ja+i-1)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$; τuq is stored in $\tau uq[ja+i-2]$ and τaup in $\tau aup[ia+i-2]$. The elements of the vectors v and u together form the m -by- nb matrix V and the nb -by- n matrix U' which are necessary, with X and Y , to apply the transformation to the unreduced part of the matrix, using a block update of the form: $\text{sub}(A) := \text{sub}(A) - V*Y' - X*U'$. The contents of $\text{sub}(A)$ on exit are illustrated by the following examples with $nb = 2$:

$m = 6$ and $n = 5 (m > n)$:

$$\begin{bmatrix} 1 & 1 & u1 & u1 & u1 \\ v1 & 1 & 1 & u2 & u2 \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

$m = 5$ and $n = 6 (m < n)$:

$$\begin{bmatrix} 1 & u1 & u1 & u1 & u1 & u1 \\ 1 & 1 & u2 & u2 & u2 & u2 \\ v1 & 1 & a & a & a & a \\ v1 & v2 & a & a & a & a \\ v1 & v2 & a & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix which is unchanged, v_i denotes an element of the vector defining $H(i)$, and u_i an element of the vector defining $G(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lacon

Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.

Syntax

```
void pslacon (MKL_INT *n , float *v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , float
*x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *isgn , float *est , MKL_INT
*kase );
```

```
void pdlacon (MKL_INT *n , double *v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv ,
double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *isgn , double *est ,
MKL_INT *kase );
```

```
void pclacon (MKL_INT *n , MKL_Complex8 *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *est ,
MKL_INT *kase );
```

```
void pzlacon (MKL_INT *n , MKL_Complex16 *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *est ,
MKL_INT *kase );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?lacon` function estimates the 1-norm of a square, real/unitary distributed matrix A . Reverse communication is used for evaluating matrix-vector products. x and v are aligned with the distributed matrix A , this information is implicitly contained within iv , ix , $descv$, and $descx$.

Input Parameters

n	(global) The length of the distributed vectors v and x . $n \geq 0$.
v	(local). Pointer into the local memory to an array of size $LOCr(n+\text{mod}(iv-1, mb_v))$. On the final return, $v = a*w$, where $est = \text{norm}(v)/\text{norm}(w)$ (w is not returned).
iv, jv	(global) The row and column indices in the global matrix V indicating the first row and the first column of the submatrix V , respectively.
$descv$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix V .
x	(local). Pointer into the local memory to an array of size $LOCr(n+\text{mod}(ix-1, mb_x))$.
ix, jx	(global) The row and column indices in the global matrix X indicating the first row and the first column of the submatrix X , respectively.
$descx$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix X .
$isgn$	(local). Array of size $LOCr(n+\text{mod}(ix-1, mb_x))$. $isgn$ is aligned with x and v .
$kase$	(local). On the initial call to <code>p?lacon</code> , $kase$ should be 0.

Output Parameters

x	(local). On an intermediate return, X should be overwritten by $A*X$, if $kase=1$, $A'*X$, if $kase=2$, <code>p?lacon</code> must be re-called with all the other parameters unchanged.
est	(global).
$kase$	(local) On an intermediate return, $kase$ is 1 or 2, indicating whether X should be overwritten by $A*X$, or $A'*X$. On the final return from <code>p?lacon</code> , $kase$ is again 0.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laconsb

Looks for two consecutive small subdiagonal elements.

Syntax

```
void pslaconsb (const float *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT
*l, MKL_INT *m, const float *h44, const float *h33, const float *h43h34, float *buf,
const MKL_INT *lwork );

void pdlaconsb (const double *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT
*l, MKL_INT *m, const double *h44, const double *h33, const double *h43h34, double *buf,
const MKL_INT *lwork );

void pclaconsb (const MKL_Complex8 *a , const MKL_INT *desca , const MKL_INT *i , const
MKL_INT *l , MKL_INT *m , const MKL_Complex8 *h44 , const MKL_Complex8 *h33 , const
MKL_Complex8 *h43h34 , MKL_Complex8 *buf , const MKL_INT *lwork );

void pzlaconsb (const MKL_Complex16 *a , const MKL_INT *desca , const MKL_INT *i ,
const MKL_INT *l , MKL_INT *m , const MKL_Complex16 *h44 , const MKL_Complex16 *h33 ,
const MKL_Complex16 *h43h34 , MKL_Complex16 *buf , const MKL_INT *lwork );
```

Include Files

- mkl_scalapack.h

Description

The `p?laconsb` function looks for two consecutive small subdiagonal elements by analyzing the effect of starting a double shift *QR* iteration given by *h44*, *h33*, and *h43h34* to see if this process makes a subdiagonal negligible.

Input Parameters

<i>a</i>	(local) Array of size <i>lld_a*LOCc(n_a)</i> . On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.
<i>desca</i>	(global and local) Array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>i</i>	(global) The global location of the bottom of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>l</i>	(global) The global location of the top of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>h44, h33, h43h34</i>	(global). These three values are for the double shift <i>QR</i> iteration.
<i>lwork</i>	(local) This must be at least $7 * \text{ceil}(\text{ceil}((i-1)/mb_a)/lcm(nprow, npcol))$. Here <i>lcm</i> is the least common multiple and <i>nprow*ncol</i> is the logical grid size.

Output Parameters

<i>m</i>	(global). On exit, this yields the starting location of the QR double shift. This will satisfy: $l \leq m \leq i-2.$
<i>buf</i>	(local). Array of size <i>lwork</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lapc2

Copies all or part of a distributed matrix to another distributed matrix.

Syntax

```
void pslapc2 (char *uplo , MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

void pdlapc2 (char *uplo , MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

void pclapc2 (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb );

void pzlapc2 (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb );
```

Include Files

- mkl_scalapack.h

Description

The p?lapc2 function copies all or part of a distributed matrix *A* to another distributed matrix *B*. No communication is performed, p?lapc2 performs a local copy $\text{sub}(A) := \text{sub}(B)$, where $\text{sub}(A)$ denotes $A(ia:ia+m-1, a:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+m-1, jb:jb+n-1)$.

p?lapc2 requires that only dimension of the matrix operands is distributed.

Input Parameters

<i>uplo</i>	(global) Specifies the part of the distributed matrix $\text{sub}(A)$ to be copied: = 'U': Upper triangular part is copied; the strictly lower triangular part of $\text{sub}(A)$ is not referenced; = 'L': Lower triangular part is copied; the strictly upper triangular part of $\text{sub}(A)$ is not referenced. Otherwise: all of the matrix $\text{sub}(A)$ is copied.
<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).

n	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
a	(local). Pointer into the local memory to an array of size $\text{lld_a} * \text{LOCc}(ja+n-1)$. On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of $\text{sub}(A)$, respectively.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
ib, jb	(global) The row and column indices in the global matrix B indicating the first row and the first column of $\text{sub}(B)$, respectively.
$descb$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix B .

Output Parameters

b	(local). Pointer into the local memory to an array of size $\text{lld_b} * \text{LOCc}(jb+n-1)$. This array contains on exit the local pieces of the distributed matrix $\text{sub}(B)$ set as follows: if $uplo = 'U'$, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq j$, $1 \leq j \leq n$; if $uplo = 'L'$, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $j \leq i \leq m$, $1 \leq j \leq n$; otherwise, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq m$, $1 \leq j \leq n$.
-----	---

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lacp3

Copies from a global parallel array into a local replicated array or vice versa.

Syntax

```
void pslacp3 (const MKL_INT *m, const MKL_INT *i, float *a, const MKL_INT *desca, float
*b, const MKL_INT *ldb, const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *rev );

void pdlacp3 (const MKL_INT *m, const MKL_INT *i, double *a, const MKL_INT *desca,
double *b, const MKL_INT *ldb, const MKL_INT *ii, const MKL_INT *jj, const MKL_INT
*rev );

void pclacp3 (const MKL_INT *m, const MKL_INT *i, MKL_Complex8 *a, const MKL_INT
*desca, MKL_Complex8 *b, const MKL_INT *ldb, const MKL_INT *ii, const MKL_INT *jj,
const MKL_INT *rev);

void pzlapcp3 (const MKL_INT *m, const MKL_INT *i, MKL_Complex16 *a, const MKL_INT
*desca, MKL_Complex16 *b, const MKL_INT *ldb, const MKL_INT *ii, const MKL_INT *jj,
const MKL_INT *rev);
```

Include Files

- `mkl_scalapack.h`

Description

This is an auxiliary function that copies from a global parallel array into a local replicated array or vice versa. Note that the entire submatrix that is copied gets placed on one node or more. The receiving node can be specified precisely, or all nodes can receive, or just one row or column of nodes.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

m	(global) m is the order of the square submatrix that is copied. $m \geq 0$. Unchanged on exit.
i	(global) The matrix element $A(i, i)$ is the global location that the copying starts from. Unchanged on exit.
a	(local) Array of size $lld_a * LOCc(n_a)$. On entry, the parallel matrix to be copied into or from.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
b	(local) Array of size $ldb * LOCc(m)$. If $rev = 0$, this is the global portion of the matrix $A(i:i+m-1, i:i+m-1)$. If $rev = 1$, this is unchanged on exit.
ldb	(local) The leading dimension of B .
ii, jj	(global) By using rev 0 and 1, data can be sent out and returned again. If $rev = 0$, then ii is destination row index and jj is destination column index for the node(s) receiving the replicated matrix B . If $ii \geq 0, jj \geq 0$, then node (ii, jj) receives the data. If $ii = -1, jj \geq 0$, then all rows in column jj receive the data. If $ii \geq 0, jj = -1$, then all cols in row ii receive the data. If $ii = -1, jj = -1$, then all nodes receive the data. If $rev \neq 0$, then ii is the source row index for the node(s) sending the replicated B .
rev	(global) Use $rev = 0$ to send global matrix A into locally replicated matrix B (on node (ii, jj)). Use $rev \neq 0$ to send locally replicated B from node (ii, jj) to its owner (which changes depending on its location in A) into the global A .

Output Parameters

<i>a</i>	On exit, if <i>rev</i> = 1, the copied data. Unchanged on exit if <i>rev</i> = 0.
<i>b</i>	If <i>rev</i> = 1, this is unchanged on exit.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
void pslacpy (char *uplo , MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

void pdlacpy (char *uplo , MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

void pclacpy (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb );

void pzlacpy (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb );
```

Include Files

- mkl_scalapack.h

Description

The `p?lacpy` function copies all or part of a distributed matrix *A* to another distributed matrix *B*. No communication is performed, `p?lacpy` performs a local copy $\text{sub}(B) := \text{sub}(A)$, where $\text{sub}(A)$ denotes $A(ia:ia+m-1, ja:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+m-1, jb:jb+n-1)$.

Input Parameters

<i>uplo</i>	(global) Specifies the part of the distributed matrix $\text{sub}(A)$ to be copied: = 'U': Upper triangular part; the strictly lower triangular part of $\text{sub}(A)$ is not referenced; = 'L': Lower triangular part; the strictly upper triangular part of $\text{sub}(A)$ is not referenced. Otherwise: all of the matrix $\text{sub}(A)$ is copied.
<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). Pointer into the local memory to an array of size <code>lld_a * LOCC(ja+n-1)</code> .

On entry, this array contains the local pieces of the distributed matrix $\text{sub}(A)$.

ia, ja

(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

desca

(global and local) array of size *dlen_*. The array descriptor for the distributed matrix A .

ib, jb

(global) The row and column indices in the global matrix B indicating the first row and the first column of $\text{sub}(B)$ respectively.

descb

(global and local) array of size *dlen_*. The array descriptor for the distributed matrix A .

Output Parameters

b

(local).

Pointer into the local memory to an array of size $\text{lld_b} * \text{LOCc}(jb+n-1)$. This array contains on exit the local pieces of the distributed matrix $\text{sub}(B)$ set as follows:

if *uplo* = 'U', $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq j$, $1 \leq j \leq n$;

if *uplo* = 'L', $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $j \leq i \leq m$, $1 \leq j \leq n$;

otherwise, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq m$, $1 \leq j \leq n$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laevswp

Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.

Syntax

```
void pslaevswp (MKL_INT *n , float *zin , MKL_INT *ldzi , float *z , MKL_INT *iz ,
MKL_INT *jz , MKL_INT *descz , MKL_INT *nvs , MKL_INT *key , float *work , MKL_INT
*lwork );
```

```
void pdlaevswp (MKL_INT *n , double *zin , MKL_INT *ldzi , double *z , MKL_INT *iz ,
MKL_INT *jz , MKL_INT *descz , MKL_INT *nvs , MKL_INT *key , double *work , MKL_INT
*lwork );
```

```
void pclaevswp (MKL_INT *n , float *zin , MKL_INT *ldzi , MKL_Complex8 *z , MKL_INT
*iz , MKL_INT *jz , MKL_INT *descz , MKL_INT *nvs , MKL_INT *key , float *rwork ,
MKL_INT *lrwork );
```

```
void pzlaevswp (MKL_INT *n , double *zin , MKL_INT *ldzi , MKL_Complex16 *z , MKL_INT
*iz , MKL_INT *jz , MKL_INT *descz , MKL_INT *nvs , MKL_INT *key , double *rwork ,
MKL_INT *lrwork );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?laevswp` function moves the eigenvectors (potentially unsorted) from where they are computed, to a ScaLAPACK standard block cyclic array, sorted so that the corresponding eigenvalues are sorted.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

n	(global) The order of the matrix A . $n \geq 0$.
zin	(local). Array of size $ldzi * nvs[iam+1]$. The eigenvectors on input. iam is a process rank from $[0, nprocs)$ interval. Each eigenvector resides entirely in one process. Each process holds a contiguous set of $nvs[iam+1]$ eigenvectors. The global number of the first eigenvector that the process holds is: ((sum for $i=[0, iam]$ of $nvs[i]$)+1).
$ldzi$	(local) The leading dimension of the zin array.
iz, jz	(global) The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z , respectively.
$descz$	(global and local) Array of size $dlen_$. The array descriptor for the distributed matrix Z .
nvs	(global) Array of size $nprocs+1$ $nvs[i]$ = number of eigenvectors held by processes $[0, i)$ $nvs[0]$ = number of eigenvectors held by processes $[0, 0) = 0$ $nvs[nprocs]$ = number of eigenvectors held by $[0, nprocs)$ = total number of eigenvectors.
key	(global) Array of size n . Indicates the actual index (after sorting) for each of the eigenvectors.
$rwork$	(local). Array of size $lrwork$.
$lrwork$	(local) Size of $work$.

Output Parameters

z	(local). Array of global size $n * n$ and of local size $lld_z * nq$. The eigenvectors on output. The eigenvectors are distributed in a block cyclic manner in both dimensions, with a block size of nb .
-----	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lahrd

Reduces the first nb columns of a general rectangular matrix A so that elements below the k -th subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A .

Syntax

```
void pslahrd (MKL_INT *n , MKL_INT *k , MKL_INT *nb , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *t , float *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , float *work );

void pdlahrd (MKL_INT *n , MKL_INT *k , MKL_INT *nb , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *t , double *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , double *work );

void pclahrd (MKL_INT *n , MKL_INT *k , MKL_INT *nb , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *t , MKL_Complex8 *y ,
MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_Complex8 *work );

void pzlahrd (MKL_INT *n , MKL_INT *k , MKL_INT *nb , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *t , MKL_Complex16
*y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description

The `p?lahrd` function reduces the first nb columns of a real general n -by- $(n-k+1)$ distributed matrix $A(ia:ia+n-1, ja:ja+n-k)$ so that elements below the k -th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation Q^*A^*Q . The function returns the matrices V and T which determine Q as a block reflector $I-V^*T^*V'$, and also the matrix $Y = A^*V^*T$.

This is an auxiliary function called by `p?gehrd`. In the following comments $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

n	(global) The order of the distributed matrix $\text{sub}(A)$. $n \geq 0$.
k	(global) The offset for the reduction. Elements below the k -th subdiagonal in the first nb columns are reduced to zero.
nb	(global) The number of columns to be reduced.
a	(local).

Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-k)$. On entry, this array contains the local pieces of the n -by- $(n-k+1)$ general distributed matrix $A(ia:ia+n-1, ja:ja+n-k)$.

<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>iy, jy</i>	(global) The row and column indices in the global matrix Y indicating the first row and the first column of the matrix sub(Y), respectively.
<i>descy</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix Y .
<i>work</i>	(local). Array of size nb .

Output Parameters

<i>a</i>	(local). On exit, the elements on and above the k -th subdiagonal in the first nb columns are overwritten with the corresponding elements of the reduced distributed matrix; the elements below the k -th subdiagonal, with the array <i>tau</i> , represent the matrix Q as a product of elementary reflectors. The other columns of the matrix $A(ia:ia+n-1, ja:ja+n-k)$ are unchanged. (See <i>Application Notes</i> below.)
<i>tau</i>	(local) Array of size $LOCc(ja+n-2)$. The scalar factors of the elementary reflectors (see <i>Application Notes</i> below). <i>tau</i> is tied to the distributed matrix A .
<i>t</i>	(local) Array of size $nb_a * nb_a$. The upper triangular matrix T .
<i>y</i>	(local). Pointer into the local memory to an array of size $lld_y * nb_a$. On exit, this array contains the local pieces of the n -by- nb distributed matrix Y . $lld_y \geq LOCr(ia+n-1)$.

Application Notes

The matrix Q is represented as a product of nb elementary reflectors

$$Q = H(1) * H(2) * \dots * H(nb).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v',$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i+k-1) = 0$, $v(i+k) = 1$; $v(i+k+1:n)$ is stored on exit in $A(ia+i+k:ia+n-1, ja+i-1)$, and τ in $\tau[ja+i-2]$.

The elements of the vectors v together form the $(n-k+1)$ -by- nb matrix V which is needed, with T and Y , to apply the transformation to the unreduced part of the matrix, using an update of the form: $A(ia:ia+n-1, ja:ja+n-k) := (I-V*T*V)*(A(ia:ia+n-1, ja:ja+n-k)-Y*V')$. The contents of $A(ia:ia+n-1, ja:ja+n-k)$ on exit are illustrated by the following example with $n = 7$, $k = 3$, and $nb = 2$:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v1 & h & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix $A(ia:ia+n-1, ja:ja+n-k)$, h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laiect

Exploits IEEE arithmetic to accelerate the computations of eigenvalues.

Syntax

```
void pslaiect (float *sigma , MKL_INT *n , float *d , MKL_INT *count );
void pdlaiectb (float *sigma , MKL_INT *n , float *d , MKL_INT *count );
void pdlaiectl (float *sigma , MKL_INT *n , float *d , MKL_INT *count );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?laiect` function computes the number of negative eigenvalues of $(A - \sigma I)$. This implementation of the Sturm Sequence loop exploits IEEE arithmetic and has no conditionals in the innermost loop. The signbit for real function `pslaiect` is assumed to be bit 32. Double-precision functions `pdlaiectb` and `pdlaiectl` differ in the order of the double precision word storage and, consequently, in the signbit location. For `pdlaiectb`, the double precision word is stored in the big-endian word order and the signbit is assumed to be bit 32. For `pdlaiectl`, the double precision word is stored in the little-endian word order and the signbit is assumed to be bit 64.

This is a ScaLAPACK internal function and arguments are not checked for unreasonable values.

Input Parameters

sigma The shift. `p?laiect` finds the number of eigenvalues less than equal to *sigma*.

n The order of the tridiagonal matrix *T*. $n \geq 1$.

d Array of size $2n-1$.

On entry, this array contains the diagonals and the squares of the off-diagonal elements of the tridiagonal matrix *T*. These elements are assumed to be interleaved in memory for better cache performance. The diagonal entries of *T* are in the entries *d*[0], *d*[2], ..., *d*[2*n*-2], while the squares of the off-diagonal entries are *d*[1], *d*[3], ..., *d*[2*n*-3]. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

Output Parameters

n The count of the number of eigenvalues of *T* less than or equal to *sigma*.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lamve
Copies all or part of one two-dimensional distributed array to another.

Syntax

```
void pslamve(char* uplo, MKL_INT* m, MKL_INT* n, float* a, MKL_INT* ia, MKL_INT* ja, MKL_INT* desca, float* b, MKL_INT* ib, MKL_INT* jb, MKL_INT* descb, float* dwork);  
void pdlamve(char* uplo, MKL_INT* m, MKL_INT* n, double* a, MKL_INT* ia, MKL_INT* ja, MKL_INT* desca, double* b, MKL_INT* ib, MKL_INT* jb, MKL_INT* descb, double* dwork);
```

Include Files

- mkl_scalapack.h

Description

p?lamve copies all or part of a distributed matrix *A* to another distributed matrix *B*. There is no alignment assumptions at all except that *A* and *B* are of the same size.

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

Input Parameters

uplo (global)
Specifies the part of the distributed matrix sub(*A*) to be copied:
= 'U': Upper triangular part is copied; the strictly lower triangular part of sub(*A*) is not referenced;
= 'L': Lower triangular part is copied; the strictly upper triangular part of sub(*A*) is not referenced;

Otherwise: All of the matrix $\text{sub}(A)$ is copied.

m

(global)

The number of rows to be operated on, which is the number of rows of the distributed matrix $\text{sub}(A)$. $m \geq 0$.

n

(global)

The number of columns to be operated on, which is the number of columns of the distributed matrix $\text{sub}(A)$. $n \geq 0$.

a

(local) pointer into the local memory to an array of size $lld_a * LOC_c(ja + n - 1)$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$ to be copied from.

ia

(global)

The row index in the global matrix A indicating the first row of $\text{sub}(A)$.

ja

(global)

The column index in the global matrix A indicating the first column of $\text{sub}(A)$.

$desca$

(global and local) array of size $dlen_$.

The array descriptor for the distributed matrix A .

ib

(global)

The row index in the global matrix B indicating the first row of $\text{sub}(B)$.

jb

(global)

The column index in the global matrix B indicating the first column of $\text{sub}(B)$.

$descb$

(global and local) array of size $dlen_$.

The array descriptor for the distributed matrix B .

$dwork$

(local workspace) array

If $uplo = 'U'$ or $uplo = 'L'$ and number of processors > 1 , the length of $dwork$ is at least as large as the length of b .

Otherwise, $dwork$ is not referenced.

OUTPUT Parameters

b

(local) pointer into the local memory to an array of size $lld_b * LOC_c(jb + n - 1)$. This array contains on exit the local pieces of the distributed matrix $\text{sub}(B)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.

Syntax

```
float pslange (char *norm , MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *work );

double pdlange (char *norm , MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *work );

float pclange (char *norm , MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *work );

double pzlange (char *norm , MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *work );
```

Include Files

- mkl_scalapack.h

Description

The `p?lange` function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

<i>norm</i>	(global) Specifies what value is returned by the function: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A , it s not a matrix norm. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$. When $m = 0$, <code>p?lange</code> is set to zero. $m \geq 0$.
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$. When $n = 0$, <code>p?lange</code> is set to zero. $n \geq 0$.
<i>a</i>	(local). Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$ containing the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>work</i>	(local). Array size $lwork$.

```

 $lwork \geq 0$  if  $norm = 'M'$  or  $'m'$  (not referenced),
 $nq0$  if  $norm = '1', 'O'$  or  $'o'$ ,
 $mp0$  if  $norm = 'I'$  or  $'i'$ ,
 $0$  if  $norm = 'F', 'f', 'E'$  or  $'e'$  (not referenced),
where
 $iroffa = \text{mod}(ia-1, mb\_a)$ ,  $icoffa = \text{mod}(ja-1, nb\_a)$ ,
 $iarow = \text{indxg2p}(ia, mb\_a, myrow, rsrc\_a, nprow)$ ,
 $iacol = \text{indxg2p}(ja, nb\_a, mycol, csrc\_a, npcol)$ ,
 $mp0 = \text{numroc}(m+iroffa, mb\_a, myrow, iarow, nprow)$ ,
 $nq0 = \text{numroc}(n+icoffa, nb\_a, mycol, iacol, npcol)$ ,
 $\text{indxg2p}$  and  $\text{numroc}$  are ScaLAPACK tool functions;  $myrow, mycol, nprow$ ,
and  $npcol$  can be determined by calling the function blacs_gridinfo.

```

Output Parameters

`val` The value returned by the function.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.

Syntax

```

float pslanhs (char *norm , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *work );

double pdlanhs (char *norm , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *work );

float pclanhs (char *norm , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *work );

double pzlanhs (char *norm , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *work );

```

Include Files

- `mk1_scalapack.h`

Description

The `p?lanhs` function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an upper Hessenberg distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

`norm` Specifies the value to be returned by the function:

= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .

= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),

= 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),

= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).

n (global)
The number of columns in the distributed matrix $\text{sub}(A)$. When $n = 0$, `p?lanhs` is set to zero. $n \geq 0$.

a (local).
Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$ containing the local pieces of the distributed matrix $\text{sub}(A)$.

ia, ja (global)
The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

desca (global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

work (local).
Array of size $lwork$.
 $lwork \geq 0$ if $norm = 'M'$ or $'m'$ (not referenced),
 $nq0$ if $norm = '1', 'O'$ or $'o'$,
 $mp0$ if $norm = 'I'$ or $'i'$,
0 if $norm = 'F', 'f', 'E'$ or $'e'$ (not referenced),
where
 $iroffa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$,
 $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$,
 $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcrow)$,
 $mp0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow)$,
 $nq0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcrow)$,
 indxg2p and numroc are ScaLAPACK tool functions; $myrow$, $mycol$, $nprow$, and $npcol$ can be determined by calling the function `blacs_gridinfo`.

Output Parameters

val The value returned by the function.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lansy, p?lanhe

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric or a complex Hermitian matrix.

Syntax

```
float pslansy (char *norm , char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *work );

double pdlansy (char *norm , char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *work );

float pclansy (char *norm , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *work );

double pzlansy (char *norm , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *work );

float pplanhe (char *norm , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *work );

double pzlanhe (char *norm , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *work );
```

Include Files

- mkl_scalapack.h

Description

The p?lansy and p?lanhe functions return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

<i>norm</i>	(global) Specifies what value is returned by the function: = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A , it s not a matrix norm. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the symmetric matrix $\text{sub}(A)$ is to be referenced. = 'U': Upper triangular part of $\text{sub}(A)$ is referenced, = 'L': Lower triangular part of $\text{sub}(A)$ is referenced.
<i>n</i>	(global)

The number of columns in the distributed matrix $\text{sub}(A)$. When $n = 0$, `p?lansy` is set to zero. $n \geq 0$.

a

(local).

Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$ containing the local pieces of the distributed matrix $\text{sub}(A)$.

If `uplo = 'U'`, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular matrix whose norm is to be computed, and the strictly lower triangular part of this matrix is not referenced. If `uplo = 'L'`, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular matrix whose norm is to be computed, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

ia, ja

(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

desca

(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

work

(local).

Array of size $lwork$.

$lwork \geq 0$ if `norm = 'M'` or `'m'` (not referenced),

$2*nq0+mp0+ldw$ if `norm = '1', 'O' or 'o', 'I' or 'i'`,

where ldw is given by:

```
if( nprow $\neq$ npcol ) then
```

```
ldw = mb_a*iceil(iceil(np0,mb_a), (lcm/nprow))
```

```
else
```

```
ldw = 0
```

```
end if
```

0 if `norm = 'F', 'f', 'E' or 'e'` (not referenced),

where lcm is the least common multiple of $nprow$ and $npcol$, $lcm = \text{ilcm}(nprow, npcol)$ and $\text{iceil}(x,y)$ is a ScaLAPACK function that returns ceiling (x/y) .

```
iroffa = mod(ia-1, mb_a), icoffa = mod(ja-1, nb_a),
```

```
iarow = indxg2p(ia, mb_a, myrow, rsrc_a, nprow),
```

```
iacol = indxg2p(ja, nb_a, mycol, csrc_a, npcol),
```

```
mp0 = numroc(m+iroffa, mb_a, myrow, iarow, nprow),
```

```
nq0 = numroc(n+icoffa, nb_a, mycol, iacol, npcol),
```

`ilcm`, `iceil`, `indxg2p`, and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the function `blacs_gridinfo`.

Output Parameters

val

The value returned by the function.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lantr

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.

Syntax

```
float pslantr (char *norm , char *uplo , char *diag , MKL_INT *m , MKL_INT *n , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *work );

double pdlantr (char *norm , char *uplo , char *diag , MKL_INT *m , MKL_INT *n , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *work );

float pclantr (char *norm , char *uplo , char *diag , MKL_INT *m , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *work );

double pzlantr (char *norm , char *uplo , char *diag , MKL_INT *m , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *work );
```

Include Files

- mkl_scalapack.h

Description

The `p?lantr` function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

<i>norm</i>	<p>(global) Specifies what value is returned by the function:</p> <ul style="list-style-type: none"> = 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A, it is not a matrix norm. = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	<p>(global)</p> <p>Specifies whether the upper or lower triangular part of the symmetric matrix $\text{sub}(A)$ is to be referenced.</p> <ul style="list-style-type: none"> = 'U': Upper trapezoidal, = 'L': Lower trapezoidal. <p>Note that $\text{sub}(A)$ is triangular instead of trapezoidal if $m = n$.</p>
<i>diag</i>	<p>(global)</p> <p>Specifies whether the distributed matrix $\text{sub}(A)$ has unit diagonal.</p>

	= 'N': Non-unit diagonal.
	= 'U': Unit diagonal.
<i>m</i>	(global) The number of rows in the distributed matrix sub(A). When $m = 0$, <code>p?lantr</code> is set to zero. $m \geq 0$.
<i>n</i>	(global) The number of columns in the distributed matrix sub(A). When $n = 0$, <code>p?lantr</code> is set to zero. $n \geq 0$.
<i>a</i>	(local). Pointer into the local memory to an array of size <code>lld_a * LOcc(ja+n-1)</code> containing the local pieces of the distributed matrix sub(A).
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.
<i>desca</i>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix A.
<i>work</i>	(local). Array size <code>lwork</code> . $lwork \geq 0$ if <code>norm</code> = 'M' or 'm' (not referenced), $nq0$ if <code>norm</code> = '1', 'O' or 'o', $mp0$ if <code>norm</code> = 'I' or 'i', 0 if <code>norm</code> = 'F', 'f', 'E' or 'e' (not referenced), $iroffa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$, $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcot)$, $mp0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow)$, $nq0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcot)$, <code>indxg2p</code> and <code>numroc</code> are ScaLAPACK tool functions; <code>myrow</code> , <code>mycol</code> , <code>nprow</code> , and <code>npcot</code> can be determined by calling the function <code>blacs_gridinfo</code> .

Output Parameters

<i>val</i>	The value returned by the function.
------------	-------------------------------------

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lapiv

Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.

Syntax

```
void pslapiv (char *direc , char *rowcol , char *pivroc , MKL_INT *m , MKL_INT *n ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT *ip ,
MKL_INT *jp , MKL_INT *descip , MKL_INT *iwork );

void pdlapiv (char *direc , char *rowcol , char *pivroc , MKL_INT *m , MKL_INT *n ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT *ip ,
MKL_INT *jp , MKL_INT *descip , MKL_INT *iwork );

void pclapiv (char *direc , char *rowcol , char *pivroc , MKL_INT *m , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT
*ip , MKL_INT *jp , MKL_INT *descip , MKL_INT *iwork );

void pzlapiv (char *direc , char *rowcol , char *pivroc , MKL_INT *m , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT
*ip , MKL_INT *jp , MKL_INT *descip , MKL_INT *iwork );
```

Include Files

- mkl_scalapack.h

Description

The `p?lapiv` function applies either P (permutation matrix indicated by *ipiv*) or $\text{inv}(P)$ to a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$, resulting in row or column pivoting. The pivot vector may be distributed across a process row or a column. The pivot vector should be aligned with the distributed matrix A . This function will transpose the pivot vector, if necessary.

For example, if the row pivots should be applied to the columns of $\text{sub}(A)$, pass `rowcol='C'` and `pivroc='C'`.

Input Parameters

<i>direc</i>	(global) Specifies in which order the permutation is applied: = 'F' (Forward): Applies pivots forward from top of matrix. Computes $P * \text{sub}(A)$. = 'B' (Backward): Applies pivots backward from bottom of matrix. Computes $\text{inv}(P) * \text{sub}(A)$.
<i>rowcol</i>	(global) Specifies if the rows or columns are to be permuted: = 'R': Rows will be permuted, = 'C': Columns will be permuted.
<i>pivroc</i>	(global) Specifies whether <i>ipiv</i> is distributed over a process row or column: = 'R': <i>ipiv</i> is distributed over a process row, = 'C': <i>ipiv</i> is distributed over a process column.
<i>m</i>	(global)

	The number of rows in the distributed matrix $\text{sub}(A)$. When $m = 0$, <code>p?lapiv</code> is set to zero. $m \geq 0$.
<code>n</code>	(global) The number of columns in the distributed matrix $\text{sub}(A)$. When $n = 0$, <code>p?lapiv</code> is set to zero. $n \geq 0$.
<code>a</code>	(local). Pointer into the local memory to an array of size <code>lld_a * LOCc(ja+n-1)</code> containing the local pieces of the distributed matrix $\text{sub}(A)$.
<code>ia, ja</code>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
<code>desca</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .
<code>ipiv</code>	(local) Array of size <code>lipiv</code> ; when <code>rowcol='R' or 'r'</code> : <code>lipiv ≥ LOCr(ia+m-1) + mb_a</code> if <code>pivroc='C' or 'c'</code> , <code>lipiv ≥ LOCc(m + mod(jp-1, nb_p))</code> if <code>pivroc='R' or 'r'</code> , and, when <code>rowcol='C' or 'c'</code> : <code>lipiv ≥ LOCr(n + mod(ip-1, mb_p))</code> if <code>pivroc='C' or 'c'</code> , <code>lipiv ≥ LOCc(ja+n-1) + nb_a</code> if <code>pivroc='R' or 'r'</code> . This array contains the pivoting information. <code>ipiv(i)</code> is the global row (column), local row (column) i was swapped with. When <code>rowcol='R' or 'r'</code> and <code>pivroc='C' or 'c'</code> , or <code>rowcol='C' or 'c'</code> and <code>pivroc='R' or 'r'</code> , the last piece of this array of size <code>mb_a</code> (resp. <code>nb_a</code>) is used as workspace. In those cases, this array is tied to the distributed matrix A .
<code>ip, jp</code>	(global) The row and column indices in the global matrix P indicating the first row and the first column of the matrix $\text{sub}(P)$, respectively.
<code>descip</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed vector <code>ipiv</code> .
<code>iwork</code>	(local). Array of size <code>ldw</code> , where <code>ldw</code> is equal to the workspace necessary for transposition, and the storage of the transposed <code>ipiv</code> : Let <code>lcm</code> be the least common multiple of <code>npro</code> and <code>npcol</code> .

```

if( *rowcol == 'r' && *pivroc == 'r') {
    if( npro == npcil) {
        ldw = LOCr( n_p + (*jp-1)%nb_p ) + nb_p;
    } else {
        ldw = LOCr( n_p + (*jp-1)%nb_p ) +
        nb_p * ceil( ceil(LOCc(n_p)/nb_p) / (lcm/npcil) );
    }
}

```

```

} else if( *rowcol == 'c' && *pivroc == 'c') {
    if( nprow == npcol ) {
        ldw = LOCc( m_p + (*ip-1)%mb_p ) + mb_p;
    } else {
        ldw = LOCc( m_p + (*ip-1)%mb_p ) +
        mb_p *ceil(ceil(LOCr(m_p)/mb_p) / (lcm/nprow) );
    }
} else {
    // iwork is not referenced.

```

Output Parameters

a (local).
On exit, the local pieces of the permuted distributed submatrix.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lapv2

Applies a permutation to an m -by- n distributed matrix.

Syntax

```

void pslapv2 (const char* direc, const char* rowcol, const MKL_INT* m, const MKL_INT*
n, float* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const MKL_INT*
ipiv, const MKL_INT* ip, const MKL_INT* jp, const MKL_INT* descip);

void pdlapv2 (const char* direc, const char* rowcol, const MKL_INT* m, const MKL_INT*
n, double* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const
MKL_INT* ipiv, const MKL_INT* ip, const MKL_INT* jp, const MKL_INT* descip);

void pclapv2 (const char* direc, const char* rowcol, const MKL_INT* m, const MKL_INT*
n, MKL_Complex8* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const
MKL_INT* ipiv, const MKL_INT* ip, const MKL_INT* jp, const MKL_INT* descip);

void pzlapv2 (const char* direc, const char* rowcol, const MKL_INT* m, const MKL_INT*
n, MKL_Complex16* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const
MKL_INT* ipiv, const MKL_INT* ip, const MKL_INT* jp, const MKL_INT* descip);

```

Include Files

- mkl_scalapack.h

Description

p?lapv2 applies either P (permutation matrix indicated by *ipiv*) or $\text{inv}(P)$ to an m -by- n distributed matrix sub(A) denoting $A(ia:ia+m-1,ja:ja+n-1)$, resulting in row or column pivoting. The pivot vector should be aligned with the distributed matrix A . For pivoting the rows of sub(A), *ipiv* should be distributed along a process column and replicated over all process rows. Similarly, *ipiv* should be distributed along a process row and replicated over all process columns for column pivoting.

Input Parameters

direc (global)
Specifies in which order the permutation is applied:

= 'F' (Forward) Applies pivots Forward from top of matrix. Computes $P * \text{sub}(A)$;

= 'B' (Backward) Applies pivots Backward from bottom of matrix. Computes $\text{inv}(P) * \text{sub}(A)$.

rowcol

(global)

Specifies if the rows or columns are to be permuted:

= 'R' Rows will be permuted,

= 'C' Columns will be permuted.

m

(global)

The number of rows to be operated on, i.e. the number of rows of the distributed submatrix $\text{sub}(A)$. $m \geq 0$.

n

(global)

The number of columns to be operated on, i.e. the number of columns of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.

a

Pointer into local memory to an array of size $ld_a * LOCc(j_a + n - 1)$.

On entry, this local array contains the local pieces of the distributed matrix $\text{sub}(A)$ to which the row or columns interchanges will be applied.

ia

(global)

The row index in the global array *a* indicating the first row of $\text{sub}(A)$.

ja

(global)

The column index in the global array *a* indicating the first column of $\text{sub}(A)$.

desca

(global and local)

Array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

ipiv

Array, size $\geq LOCr(m_a) + mb_a$ if *rowcol* = 'R', $LOCc(n_a) + nb_a$ otherwise.

It contains the pivoting information. *ipiv*[*i* - 1] is the global row (column), local row (column) *i* was swapped with. The last piece of the array of size *mb_a* or *nb_a* is used as workspace. *ipiv* is tied to the distributed matrix *A*.

ip

(global)

The global row index of *ipiv*, which points to the beginning of the submatrix on which to operate.

jp

(global)

The global column index of *ipiv*, which points to the beginning of the submatrix on which to operate.

descip

(global and local)

Array of size 8.

The array descriptor for the distributed matrix *ipiv*.

Output Parameters

a On exit, this array contains the local pieces of the permuted distributed matrix.

p?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ.

Syntax

```
void pslagge (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *r , float *c , float *rowcnd , float *colcnd , float *amax , char
*equed );
```

```
void pdlagge (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *r , double *c , double *rowcnd , double *colcnd , double *amax , char
*equed );
```

```
void pclagge (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *r , float *c , float *rowcnd , float *colcnd , float *amax ,
char *equed );
```

```
void pzlagge (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *r , double *c , double *rowcnd , double *colcnd , double
*amax , char *equed );
```

Include Files

- mkl_scalapack.h

Description

The p?laqge function equilibrates a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ using the row and scaling factors in the vectors r and c computed by p?geequ.

Input Parameters

m	(global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
n	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
a	(local). Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$. On entry, this array contains the distributed matrix $\text{sub}(A)$.
ia, ja	(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

<i>r</i>	(local). Array of size $LOCr(m_a)$. The row scale factors for sub(A). <i>r</i> is aligned with the distributed matrix <i>A</i> , and replicated across every process column. <i>r</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local). Array of size $LOCc(n_a)$. The row scale factors for sub(A). <i>c</i> is aligned with the distributed matrix <i>A</i> , and replicated across every process column. <i>c</i> is tied to the distributed matrix <i>A</i> .
<i>rowcnd</i>	(local). The global ratio of the smallest $r[i]$ to the largest $r[i]$, $ia-1 \leq i \leq ia+m-2$.
<i>colcnd</i>	(local). The global ratio of the smallest $c[i]$ to the largest $c[i]$, $ia-1 \leq i \leq ia+n-2$.
<i>amax</i>	(global). Absolute value of largest distributed submatrix entry.

Output Parameters

<i>a</i>	(local). On exit, the equilibrated distributed matrix. See <i>equed</i> for the form of the equilibrated distributed submatrix.
<i>equed</i>	(global) Specifies the form of equilibration that was done. = 'N': No equilibration = 'R': Row equilibration, that is, sub(A) has been pre-multiplied by $\text{diag}(r[ia-1:ia+m-2])$, = 'C': column equilibration, that is, sub(A) has been post-multiplied by $\text{diag}(c[ja-1:ja+n-2])$, = 'B': Both row and column equilibration, that is, sub(A) has been replaced by $\text{diag}(r[ia-1:ia+m-2]) * \text{sub}(A) * \text{diag}(c[ja-1:ja+n-2])$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr0

Computes the eigenvalues of a Hessenberg matrix and optionally returns the matrices from the Schur decomposition.

Syntax

```
void pslaqr0(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ilo, MKL_INT* ihi,
float* h, MKL_INT* desch, float* wr, float* wi, MKL_INT* iloz, MKL_INT* ihiz, float* z,
MKL_INT* descz, float* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT*
info, MKL_INT* reclevel);
```

```
void pdlaqr0(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ilo, MKL_INT* ihi,
double* h, MKL_INT* desch, double* wr, double* wi, MKL_INT* iloz, MKL_INT* ihiz, double*
z, MKL_INT* descz, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork,
MKL_INT* info, MKL_INT* recllevel);
```

Include Files

- mkl_scalapack.h

Description

`p?laqr0` computes the eigenvalues of a Hessenberg matrix H and, optionally, the matrices T and Z from the Schur decomposition $H = Z^* T^* Z^T$, where T is an upper quasi-triangular matrix (the Schur form), and Z is the orthogonal matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal matrix Q so that this function can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal matrix Q : $A = Q * H * Q^T = (QZ) * T * (QZ)^T$.

Input Parameters

<i>wantt</i>	(global) Non-zero : the full Schur form T is required; Zero : only eigenvalues are required.
<i>wantz</i>	(global) Non-zero : the matrix of Schur vectors Z is required; Zero: Schur vectors are not required.
<i>n</i>	(global) The order of the Hessenberg matrix H (and Z if <i>wantz</i> is non-zero). $n \geq 0$.
<i>ilo, ihi</i>	(global) It is assumed that the matrix H is already upper triangular in rows and columns $1:ilo-1$ and $ihiz+1:n$. <i>ilo</i> and <i>ihiz</i> are normally set by a previous call to <code>p?gebal</code> , and then passed to <code>p?gehrd</code> when the matrix output by <i>ihiz</i> is reduced to Hessenberg form. Otherwise <i>ilo</i> and <i>ihiz</i> should be set to 1 and n , respectively. If $n > 0$, then $1 \leq ilo \leq ihiz \leq n$. If $n = 0$, then <i>ilo</i> = 1 and <i>ihiz</i> = 0.
<i>h</i>	(global) array of size $lld_h * LOC_c(n)$ The upper Hessenberg matrix H .
<i>desch</i>	(global and local) Array of size $dlen_$. The array descriptor for the distributed matrix H .
<i>iloiz, ihiz</i>	Specify the rows of the matrix Z to which transformations must be applied if <i>wantz</i> is non-zero, $1 \leq iloiz \leq ilo$; $ihiz \leq ihiz \leq n$.
<i>z</i>	Array of size $lld_z * LOC_c(n)$. If <i>wantz</i> is non-zero, contains the matrix Z .

	If <i>wantzequals</i> zero, <i>z</i> is not referenced.
<i>descz</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	(local workspace) array of size <i>lwork</i>
<i>lwork</i>	(local) The length of the workspace array <i>work</i> .
<i>iwork</i>	(local workspace) array of size <i>liwork</i>
<i>liwork</i>	(local) The length of the workspace array <i>iwork</i> .
<i>recllevel</i>	(local) Level of recursion. <i>recllevel</i> = 0 must hold on entry.

OUTPUT Parameters

<i>h</i>	On exit, if <i>wantt</i> is non-zero, the matrix <i>H</i> is upper quasi-triangular in rows and columns <i>ilo:ihi</i> , with 1-by-1 and 2-by-2 blocks on the main diagonal. The 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with $H(i,i) = H(i+1,i+1)$ and $H(i+1,i)*H(i,i+1) < 0$. If <i>info</i> = 0 and <i>wanttequals</i> zero, the contents of <i>h</i> are unspecified on exit.
<i>wr, wi</i>	The real and imaginary parts, respectively, of the computed eigenvalues <i>ilo</i> to <i>ihi</i> are stored in the corresponding elements of <i>wr</i> and <i>wi</i> . If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of <i>wr</i> and <i>wi</i> , say the <i>i</i> -th and (<i>i</i> +1)th, with $wi[i-1] > 0$ and $wi[i] < 0$. If <i>wantt</i> is non-zero, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in <i>h</i> .
<i>z</i>	Updated matrix with transformations applied only to the submatrix $Z(ilo:ihi,ilo:ihi)$. If COMPZ = 'I', on exit, if <i>info</i> = 0, <i>z</i> contains the orthogonal matrix <i>Z</i> of the Schur vectors of <i>H</i> . If <i>wantz</i> is non-zero, then $Z(ilo:ihi,iloz:ihiz)$ is replaced by $Z(ilo:ihi,iloz:ihiz)*U$, when <i>U</i> is the orthogonal/unitary Schur factor of $H(ilo:ihi,ilo:ihi)$. If <i>wantzequals</i> zero, then <i>z</i> is not defined.
<i>work</i> [0]	On exit, if <i>info</i> = 0, <i>work</i> [0] returns the optimal <i>lwork</i> .
<i>iwork</i> [0]	On exit, if <i>info</i> = 0, <i>iwork</i> [0] returns the optimal <i>liwork</i> .
<i>info</i>	> 0: if <i>info</i> = <i>i</i> , then the function failed to compute all the eigenvalues. Elements 0: <i>ilo</i> -2 and <i>i</i> : <i>n</i> -1 of <i>wr</i> and <i>wi</i> contain those eigenvalues which have been successfully computed.

> 0: if *wantte* equals zero, then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *ihi* of the final output value of *H*.

> 0: if *wantt* is non-zero, then (initial value of *H*)**U* = *U**(final value of *H*), where *U* is an orthogonal/unitary matrix. The final value of *H* is upper Hessenberg and quasi-triangular/triangular in rows and columns *info*+1 through *ihi*.

> 0: if *wantz* is non-zero, then (final value of $Z(ilo:ihi, iloz:ihiz)$)=(initial value of $Z(ilo:ihi, iloz:ihiz)$)**U*, where *U* is the orthogonal/unitary matrix in the previous expression (regardless of the value of *wantt*).

> 0: if *wantz* equals zero, then *z* is not accessed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr1

Sets a scalar multiple of the first column of the product of a 2-by-2 or 3-by-3 matrix and specified shifts.

Syntax

```
void pslaqr1(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ilo, MKL_INT* ihi,
float* a, MKL_INT* desca, float* wr, float* wi, MKL_INT* iloz, MKL_INT* ihiz, float* z,
MKL_INT* descz, float* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* ilwork, MKL_INT*
info);

void pdlaqr1(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ilo, MKL_INT* ihi,
double* a, MKL_INT* desca, double* wr, double* wi, MKL_INT* iloz, MKL_INT* ihiz, double*
z, MKL_INT* descz, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* ilwork,
MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

p?laqr1 is an auxiliary function used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns *ilo* to *ihi*.

This is a modified version of p?lahqr from ScaLAPACK version 1.7.3. The following modifications were made:

- Workspace query functionality was added.
- Aggressive early deflation is implemented.
- Aggressive deflation (looking for two consecutive small subdiagonal elements by PSLACONS) is abandoned.
- The returned Schur form is now in canonical form, i.e., the returned 2-by-2 blocks really correspond to complex conjugate pairs of eigenvalues.
- For some reason, the original version of p?lahqr sometimes did not read out the converged eigenvalues correctly. This is now fixed.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>wantt</i>	(global) Non-zero : the full Schur form T is required; Zero: only eigenvalues are required.
<i>wantz</i>	Non-zero : the matrix of Schur vectors Z is required; Zero: Schur vectors are not required.
<i>n</i>	(global) The order of the Hessenberg matrix A (and Z if <i>wantz</i> is non-zero). $n \geq 0$.
<i>ilo, ihi</i>	(global) It is assumed that the matrix A is already upper quasi-triangular in rows and columns $ihi+1:n$, and that $A(ilo, ilo-1) = 0$ (unless $ilo = 1$). <i>p?laqr1</i> works primarily with the Hessenberg submatrix in rows and columns ilo to ihi , but applies transformations to all of H if <i>wantt</i> is non-zero. $1 \leq ilo \leq \max(1, ihi); ihi \leq n$.
<i>a</i>	(global) array of size $lld_a * LOC_c(n)$ On entry, the upper Hessenberg matrix A .
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>iloz, ihiz</i>	(global) Specify the rows of the matrix Z to which transformations must be applied if <i>wantz</i> is non-zero. $1 \leq iloz \leq ilo; ihi \leq ihiz \leq n$.
<i>z</i>	(global) array of size $lld_z * LOC_c(n)$. If <i>wantz</i> is non-zero, on entry <i>z</i> must contain the current matrix Z of transformations accumulated by <i>p?hseqr</i> . If <i>wantz</i> is zero, <i>z</i> is not referenced.
<i>descz</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix Z .
<i>work</i>	(local output) array of size <i>lwork</i>
<i>lwork</i>	(local) The size of the work array ($lwork \geq 1$).

If *lwork* = -1, then a workspace query is assumed.

iwork (global and local) array of size *ilwork*

This holds the some of the IBLK integer arrays.

ilwork (local)

The size of the *iwork* array (*ilwork* ≥ 3).

OUTPUT Parameters

a If *wantt* is non-zero, the matrix *A* is upper quasi-triangular in rows and columns *ilo:ihi*, with any 2-by-2 or larger diagonal blocks not yet in standard form. If *wantt* equals zero, the contents of *a* are unspecified on exit.

wr, wi (global replicated) array of size *n*

The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ihi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)th, with *wi*[*i*-1] > 0 and *wi*[*i*] < 0. If *wantt* is non-zero, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *a*. *a* may be returned with larger diagonal blocks until the next release.

z On exit *z* is updated; transformations are applied only to the submatrix *Z*(*iloz:ihiz,ilo:ihi*).

If *wantz* equals zero, *z* is not referenced.

work[0] On exit, if *info* = 0, *work*[0] returns the optimal *lwork*.

info (global)

< 0: parameter number -*info* incorrect or inconsistent

= 0: successful exit

> 0: p?laqr1 failed to compute all the eigenvalues *ilo* to *ihi* in a total of 30*(*ihi-ilo*+1) iterations; if *info* = *i*, elements *i:ihi-1* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

Application Notes

This algorithm is very similar to p?ahqr. Unlike p?lahqr, instead of sending one double shift through the largest unreduced submatrix, this algorithm sends multiple double shifts and spaces them apart so that there can be parallelism across several processor row/columns. Another critical difference is that this algorithm aggregates multiple transforms together in order to apply them in a block fashion.

Current Notes and/or Restrictions:

- This code requires the distributed block size to be square and at least six (6); unlike simpler codes like LU, this algorithm is extremely sensitive to block size. Unwise choices of too small a block size can lead to bad performance.
- This code requires *a* and *z* to be distributed identically and have identical contexts.
- This release currently does not have a function for resolving the Schur blocks into regular 2x2 form after this code is completed. Because of this, a significant performance impact is required while the deflation is done by sometimes a single column of processors.

- This code does not currently block the initial transforms so that none of the rows or columns for any bulge are completed until all are started. To offset pipeline start-up it is recommended that at least $2 \times \text{LCM}(\text{NPROW}, \text{NPCOL})$ bulges are used (if possible)
- The maximum number of bulges currently supported is fixed at 32. In future versions this will be limited only by the incoming *work* array.
- The matrix *A* must be in upper Hessenberg form. If elements below the subdiagonal are nonzero, the resulting transforms may be nonsimilar. This is also true with the LAPACK function.
- For this release, it is assumed *rsrc_*=*csrc_*=0
- Currently, all the eigenvalues are distributed to all the nodes. Future releases will probably distribute the eigenvalues by the column partitioning.
- The internals of this function are subject to change.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr2

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
void pslaqr2(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ktop, MKL_INT* kbot,
MKL_INT* nw, float* a, MKL_INT* desca, MKL_INT* iloz, MKL_INT* ihiz, float* z, MKL_INT*
descz, MKL_INT* ns, MKL_INT* nd, float* sr, float* si, float* t, MKL_INT* ldt, float* v,
MKL_INT* ldv, float* wr, float* wi, float* work, MKL_INT* lwork);
```

```
void pdlaqr2(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ktop, MKL_INT* kbot,
MKL_INT* nw, double* a, MKL_INT* desca, MKL_INT* iloz, MKL_INT* ihiz, double* z,
MKL_INT* descz, MKL_INT* ns, MKL_INT* nd, double* sr, double* si, double* t, MKL_INT*
ldt, double* v, MKL_INT* ldv, double* wr, double* wi, double* work, MKL_INT* lwork);
```

Include Files

- `mk1_scalapack.h`

Description

`p?laqr2` accepts as input an upper Hessenberg matrix *A* and performs an orthogonal similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output *A* is overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal similarity transformation of *A*. It is to be hoped that the final version of *A* has many zero subdiagonal entries.

This function handles small deflation windows which is affordable by one processor. Normally, it is called by `p?laqr1`. All the inputs are assumed to be valid without checking.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

wantt (global)

If *wantt* is non-zero, then the Hessenberg matrix *A* is fully updated so that the quasi-triangular Schur factor may be computed (in cooperation with the calling function).

If *wantt* equals zero, then only enough of *A* is updated to preserve the eigenvalues.

wantz

(global)

If *wantz* is non-zero, then the orthogonal matrix *Z* is updated so that the orthogonal Schur factor may be computed (in cooperation with the calling function).

If *wantz* equals zero, then *z* is not referenced.

n

(global)

The order of the matrix *A* and (if *wantz* is non-zero) the order of the orthogonal matrix *Z*.

ktop, kbot

(global)

It is assumed without a check that either $kbot = n$ or $A(kbot+1, kbot)=0$. *kbot* and *ktop* together determine an isolated block along the diagonal of the Hessenberg matrix. However, $A(ktop, ktop-1)=0$ is not essentially necessary if *wantt* is non-zero .

nw

(global)

Deflation window size. $1 \leq nw \leq (kbot-ktop+1)$. Normally $nw \geq 3$ if *p?laqr2* is called by *p?laqr1*.

a

(local) array of size $lld_a * LOC_c(n)$

The initial *n*-by-*n* section of *a* stores the Hessenberg matrix undergoing aggressive early deflation.

desca

(global and local) array of size *dlen_*.

The array descriptor for the distributed matrix *A*.

iloz, ihiz

(global)

Specify the rows of the matrix *Z* to which transformations must be applied if *wantz* is non-zero. $1 \leq iloz \leq ihiz \leq n$.

z

Array of size $lld_z * LOC_c(n)$

If *wantz* is non-zero, then on output, the orthogonal similarity transformation mentioned above has been accumulated into the matrix *Z*(*iloz:ihiz*,

kbot:ktop), stored in *z*, from the right.

If *wantz* is zero, then *z* is unreferenced.

descz

(global and local) array of size *dlen_*.

The array descriptor for the distributed matrix *Z*.

t

(local workspace) array of size *ldt* * *nw*.

ldt

(local)

	The leading dimension of the array <i>t</i> . $ldt \geq nw$.
<i>v</i>	(local workspace) array of size $ldv * nw$.
<i>ldv</i>	(local)
	The leading dimension of the array <i>v</i> . $ldv \geq nw$.
<i>wr, wi</i>	(local workspace) array of size <i>kbot</i> .
<i>work</i>	(local workspace) array of size <i>lwork</i> .
<i>lwork</i>	(local)
	<i>work(lwork)</i> is a local array and <i>lwork</i> is assumed big enough so that $lwork \geq nw * nw$.

OUTPUT Parameters

<i>a</i>	On output <i>a</i> has been transformed by an orthogonal similarity transformation, perturbed, and returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.
<i>z</i>	
<i>ns</i>	(global) The number of unconverged (that is, approximate) eigenvalues returned in <i>sr</i> and <i>si</i> that may be used as shifts by the calling function.
<i>nd</i>	(global) The number of converged eigenvalues uncovered by this function.
<i>sr, si</i>	(global) array of size <i>kbot</i> On output, the real and imaginary parts of approximate eigenvalues that may be used for shifts are stored in <i>sr</i> [<i>kbot-nd-ns</i>] through <i>sr</i> [<i>kbot-nd-1</i>] and <i>si</i> [<i>kbot-nd-ns</i>] through <i>si</i> [<i>kbot-nd-1</i>], respectively. On processor #0, the real and imaginary parts of converged eigenvalues are stored in <i>sr</i> [<i>kbot-nd</i>] through <i>sr</i> [<i>kbot-1</i>] and <i>si</i> [<i>kbot-nd</i>] through <i>si</i> [<i>kbot-1</i>], respectively. On other processors, these entries are set to zero.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr3

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
void pslaqr3(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ktop, MKL_INT* kbot,
MKL_INT* nw, float* h, MKL_INT* desch, MKL_INT* iloz, MKL_INT* ihiz, float* z, MKL_INT*
descz, MKL_INT* ns, MKL_INT* nd, float* sr, float* si, float* v, MKL_INT* descv,
MKL_INT* nh, float* t, MKL_INT* desct, MKL_INT* nv, float* wv, MKL_INT* descw, float*
work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* reclevel);
```

```
void pdlaqr3(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ktop, MKL_INT* kbot,
MKL_INT* nw, double* h, MKL_INT* desch, MKL_INT* iloz, MKL_INT* ihiz, double* z,
MKL_INT* descz, MKL_INT* ns, MKL_INT* nd, double* sr, double* si, double* v, MKL_INT*
descv, MKL_INT* nh, double* t, MKL_INT* desct, MKL_INT* nv, double* wv, MKL_INT* descw,
double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* reclevel);
```

Include Files

- mkl_scalapack.h

Description

This function accepts as input an upper Hessenberg matrix H and performs an orthogonal similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output H is overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal similarity transformation of H . It is to be hoped that the final version of H has many zero subdiagonal entries.

Input Parameters

<i>wantt</i>	(global) If <i>wantt</i> is non-zero, then the Hessenberg matrix H is fully updated so that the quasi-triangular Schur factor may be computed (in cooperation with the calling function). If <i>wantt</i> equals zero, then only enough of H is updated to preserve the eigenvalues.
<i>wantz</i>	(global) If <i>wantz</i> is non-zero, then the orthogonal matrix Z is updated so that the orthogonal Schur factor may be computed (in cooperation with the calling function). If <i>wantz</i> equals zero, then z is not referenced.
<i>n</i>	(global) The order of the matrix H and (if <i>wantz</i> is non-zero), the order of the orthogonal matrix Z .
<i>ktop</i>	(global) It is assumed that either $ktop = 1$ or $H(ktop,ktop-1)=0$. <i>kbot</i> and <i>ktop</i> together determine an isolated block along the diagonal of the Hessenberg matrix.
<i>kbot</i>	(global) It is assumed without a check that either $kbot = n$ or $H(kbot+1,kbot)=0$. <i>kbot</i> and <i>ktop</i> together determine an isolated block along the diagonal of the Hessenberg matrix.
<i>nw</i>	(global) Deflation window size. $1 \leq nw \leq (kbot-ktop+1)$.
<i>h</i>	(local) array of size $lld_h * LOC_c(n)$ The initial n -by- n section of H stores the Hessenberg matrix undergoing aggressive early deflation.

<i>desch</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>H</i> .
<i>iloz, ihiz</i>	(global) Specify the rows of the matrix <i>Z</i> to which transformations must be applied if <i>wantz</i> is non-zero. $1 \leq iloz \leq ihiz \leq n$.
<i>z</i>	Array of size $lld_z * LOC_c(n)$ If <i>wantz</i> is non-zero, then on output, the orthogonal similarity transformation mentioned above has been accumulated into the matrix $Z(iloz:ihiz,kbot:ktop)$ from the right. If <i>wantz</i> is zero, then <i>z</i> is unreferenced.
<i>descz</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> .
<i>v</i>	(global workspace) array of size $lld_v * LOC_c(nw)$ An <i>nw</i> -by- <i>nw</i> distributed work array.
<i>descv</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>V</i> .
<i>nh</i>	The number of columns of <i>t</i> . $nh \geq nw$.
<i>t</i>	(global workspace) array of size $lld_t * LOC_c(nh)$
<i>desct</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>T</i> .
<i>nv</i>	(global) The number of rows of work array <i>wv</i> available for workspace. $nv \geq nw$.
<i>wv</i>	(global workspace) array of size $lld_w * LOC_c(nw)$
<i>descw</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>wv</i> .
<i>work</i>	(local workspace) array of size <i>lwork</i> .
<i>lwork</i>	(local) The size of the work array <i>work</i> ($lwork \geq 1$). $lwork = 2 * nw$ suffices, but greater efficiency may result from larger values of <i>lwork</i> . If $lwork = -1$, then a workspace query is assumed; p?laqr3 only estimates the optimal workspace size for the given values of <i>n</i> , <i>nw</i> , <i>ktop</i> and <i>kbot</i> . The estimate is returned in <i>work</i> [0]. No error message related to <i>lwork</i> is issued by xerbla. Neither <i>h</i> nor <i>z</i> are accessed.
<i>iwork</i>	(local workspace) array of size <i>liwork</i>
<i>liwork</i>	(local) The length of the workspace array <i>iwork</i> ($liwork \geq 1$).

If *liwork*=-1, then a workspace query is assumed.

OUTPUT Parameters

<i>h</i>	On output <i>h</i> has been transformed by an orthogonal similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.
<i>z</i>	IF <i>wantz</i> is non-zero, then on output, the orthogonal similarity transformation mentioned above has been accumulated into the matrix <i>Z(iloz:ihiz,kbot:ktop)</i> from the right. If <i>wantz</i> is zero, then <i>z</i> is unreferenced.
<i>ns</i>	(global) The number of unconverged (that is, approximate) eigenvalues returned in <i>sr</i> and <i>si</i> that may be used as shifts by the calling function.
<i>nd</i>	(global) The number of converged eigenvalues uncovered by this function.
<i>sr, si</i>	(global) array of size <i>kbot</i> . The real and imaginary parts of approximate eigenvalues that may be used for shifts are stored in <i>sr[kbot-nd-ns]</i> through <i>sr[kbot-nd-1]</i> and <i>si[kbot-nd-ns]</i> through <i>si[kbot-nd-1]</i> , respectively. The real and imaginary parts of converged eigenvalues are stored in <i>sr[kbot-nd]</i> through <i>sr[kbot-1]</i> and <i>si[kbot-nd]</i> through <i>si[kbot-1]</i> , respectively.
<i>work</i> [0]	On exit, if <i>info</i> = 0, <i>work</i> [0] returns the optimal <i>lwork</i>
<i>iwork</i> [0]	On exit, if <i>info</i> = 0, <i>iwork</i> [0] returns the optimal <i>liwork</i>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqr5

Performs a single small-bulge multi-shift QR sweep.

Syntax

```
void pslaqr5(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* kacc22, MKL_INT* n, MKL_INT*
ktop, MKL_INT* kbot, MKL_INT* nshfts, float* sr, float* si, float* h, MKL_INT* desch,
MKL_INT* iloz, MKL_INT* ihiz, float* z, MKL_INT* descz, float* work, MKL_INT* lwork,
MKL_INT* iwork, MKL_INT* liwork);

void pdlaqr5(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* kacc22, MKL_INT* n, MKL_INT*
ktop, MKL_INT* kbot, MKL_INT* nshfts, double* sr, double* si, double* h, MKL_INT* desch,
MKL_INT* iloz, MKL_INT* ihiz, double* z, MKL_INT* descz, double* work, MKL_INT* lwork,
MKL_INT* iwork, MKL_INT* liwork);
```

Include Files

- `mkl_scalapack.h`

Description

This auxiliary function called by `p?laqr0` performs a single small-bulge multi-shift QR sweep by chasing separated groups of bulges along the main block diagonal of a Hessenberg matrix H .

Input Parameters

<code>wantt</code>	(global) scalar <code>wantt</code> is non-zero if the quasi-triangular Schur factor is being computed. <code>wantt</code> is set to zero otherwise.
<code>wantz</code>	(global) scalar <code>wantz</code> is non-zero if the orthogonal Schur factor is being computed. <code>wantz</code> is set to zero otherwise.
<code>kacc22</code>	(global) Value 0, 1, or 2. Specifies the computation mode of far-from-diagonal orthogonal updates. = 0: <code>p?laqr5</code> does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries. = 1: <code>p?laqr5</code> accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries. = 2: <code>p?laqr5</code> accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.
<code>n</code>	(global) scalar The order of the Hessenberg matrix H and, if <code>wantz</code> is non-zero, the order of the orthogonal matrix Z .
<code>ktop, kbot</code>	(global) scalar These are the first and last rows and columns of an isolated diagonal block upon which the QR sweep is to be applied. It is assumed without a check that either $ktop = 1$ or $H(ktop, ktop-1) = 0$ and either $kbot = n$ or $H(kbot+1, kbot) = 0$.
<code>nshfts</code>	(global) scalar <code>nshfts</code> gives the number of simultaneous shifts. <code>nshfts</code> must be positive and even.
<code>sr, si</code>	(global) Array of size <code>nshfts</code> <code>sr</code> contains the real parts and <code>si</code> contains the imaginary parts of the <code>nshfts</code> shifts of origin that define the multi-shift QR sweep.
<code>h</code>	(local) Array of size <code>lld_h * LOC_c(n)</code> On input <code>h</code> contains a Hessenberg matrix H .
<code>desch</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix H .

<i>iloz, ihiz</i>	(global) Specify the rows of the matrix <i>Z</i> to which transformations must be applied if <i>wantzis</i> non-zero. $1 \leq iloz \leq ihiz \leq n$
<i>z</i>	(local) array of size $lld_z * LOC_c(n)$ If <i>wantzis</i> non-zero, then the QR Sweep orthogonal similarity transformation is accumulated into the matrix $Z(iloz:ihiz,kbot:ktop)$ from the right. If <i>wantzequals</i> zero, then <i>z</i> is unreferenced.
<i>descz</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	(local workspace) array of size <i>lwork</i>
<i>lwork</i>	(local) The size of the <i>work</i> array ($lwork \geq 1$). If <i>lwork</i> =-1, then a workspace query is assumed.
<i>iwork</i>	(local workspace) array of size <i>liwork</i>
<i>liwork</i>	(local) The size of the <i>iwork</i> array ($liwork \geq 1$). If <i>liwork</i> =-1, then a workspace query is assumed.

Output Parameters

<i>h</i>	A multi-shift QR sweep with shifts $sr(j)+i*si(j)$ is applied to the isolated diagonal block in rows and columns <i>ktop</i> through <i>kbot</i> of the matrix <i>H</i> .
<i>z</i>	If <i>wantzis</i> non-zero, <i>z</i> is updated with transformations applied only to the submatrix $Z(iloz:ihiz,kbot:ktop)$.
<i>work</i> [0]	On exit, if <i>info</i> = 0, <i>work</i> [0] returns the optimal <i>lwork</i> .
<i>iwork</i> [0]	On exit, if <i>info</i> = 0, <i>iwork</i> [0] returns the optimal <i>liwork</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ .

Syntax

```
void pslaqsy (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *sr , float *sc , float *scond , float *amax , char *equeued );

void pdlaqsy (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *sr , double *sc , double *scond , double *amax , char *equeued );

void pclaqsy (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *sr , float *sc , float *scond , float *amax , char *equeued );
```

```
void pzlaqsy (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *sr , double *sc , double *scond , double *amax , char *equed );
```

Include Files

- mkl_scalapack.h

Description

The `p?laqsy` function equilibrates a symmetric distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the scaling factors in the vectors `sr` and `sc`. The scaling factors are computed by `p?poequ`.

Input Parameters

<code>uplo</code>	<p>(global) Specifies the upper or lower triangular part of the symmetric distributed matrix <code>sub(A)</code> is to be referenced:</p> <p>= 'U': Upper triangular part;</p> <p>= 'L': Lower triangular part.</p>
<code>n</code>	<p>(global)</p> <p>The order of the distributed matrix <code>sub(A)</code>. $n \geq 0$.</p>
<code>a</code>	<p>(local).</p> <p>Pointer into the local memory to an array of size <code>lld_a * LOCc(ja+n-1)</code>.</p> <p>On entry, this array contains the local pieces of the distributed matrix <code>sub(A)</code>. On entry, the local pieces of the distributed symmetric matrix <code>sub(A)</code>.</p> <p>If <code>uplo = 'U'</code>, the leading n-by-n upper triangular part of <code>sub(A)</code> contains the upper triangular part of the matrix, and the strictly lower triangular part of <code>sub(A)</code> is not referenced.</p> <p>If <code>uplo = 'L'</code>, the leading n-by-n lower triangular part of <code>sub(A)</code> contains the lower triangular part of the matrix, and the strictly upper triangular part of <code>sub(A)</code> is not referenced.</p>
<code>ia, ja</code>	<p>(global)</p> <p>The row and column indices in the global matrix <code>A</code> indicating the first row and the first column of the matrix <code>sub(A)</code>, respectively.</p>
<code>desca</code>	<p>(global and local) array of size <code>dlen_</code>. The array descriptor for the distributed matrix <code>A</code>.</p>
<code>sr</code>	<p>(local)</p> <p>Array of size <code>LOCr(m_a)</code>. The scale factors for the matrix <code>A(ia:ia+m-1, ja:ja+n-1)</code>. <code>sr</code> is aligned with the distributed matrix <code>A</code>, and replicated across every process column. <code>sr</code> is tied to the distributed matrix <code>A</code>.</p>
<code>sc</code>	<p>(local)</p> <p>Array of size <code>LOCc(m_a)</code>. The scale factors for the matrix <code>A(ia:ia+m-1, ja:ja+n-1)</code>. <code>sc</code> is aligned with the distributed matrix <code>A</code>, and replicated across every process column. <code>sc</code> is tied to the distributed matrix <code>A</code>.</p>
<code>scond</code>	<p>(global).</p>

Ratio of the smallest $sr[i]$ (respectively $sc[j]$) to the largest $sr[i]$ (respectively $sc[j]$), with $ia - 1 \leq i < ia + n - 1$ and $ja - 1 \leq j < ja + n - 1$.

amax

(global).

Absolute value of largest distributed submatrix entry.

Output Parameters

a

On exit,

if *equed* = 'Y', the equilibrated matrix:

$\text{diag}(sr_{ia}, \dots, sr_{ia+n-1}) * \text{sub}(A) * \text{diag}(sc_{ja}, \dots, sc_{ja+n-1})$.

equed

(global).

Specifies whether or not equilibration was done.

= 'N': No equilibration.

= 'Y': Equilibration was done, that is, $\text{sub}(A)$ has been replaced by:

$\text{diag}(sr_{ia}, \dots, sr_{ia+n-1}) * \text{sub}(A) * \text{diag}(sc_{ja}, \dots, sc_{ja+n-1})$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lared1d

Redistributes an array assuming that the input array, bycol, is distributed across rows and that all process columns contain the same copy of bycol.

Syntax

```
void pslared1d (MKL_INT *n , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , float *bycol ,
float *byall , float *work , MKL_INT *lwork );
```

```
void pdlared1d (MKL_INT *n , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , double
*bycol , double *byall , double *work , MKL_INT *lwork );
```

Include Files

- mkl_scalapack.h

Description

The `p?lared1d` function redistributes a 1D array. It assumes that the input array *bycol* is distributed across rows and that all process column contain the same copy of *bycol*. The output array *byall* is identical on all processes and contains the entire array.

Input Parameters

np = Number of local rows in *bycol*()

n

(global)

The number of elements to be redistributed. $n \geq 0$.

ia, ja

(global) *ia, ja* must be equal to 1.

desc

(local) array of size 9. A 2D array descriptor, which describes *bycol*.

<i>bycol</i>	(local). Distributed block cyclic array of global size <i>n</i> and of local size <i>np</i> . <i>bycol</i> is distributed across the process rows. All process columns are assumed to contain the same value.
<i>work</i>	(local). size <i>lwork</i> . Used to hold the buffers sent from one process to another.
<i>lwork</i>	(local) The size of the <i>work</i> array. $lwork \geq \text{numroc}(n, \text{desc}[nb_], 0, 0, npcol)$.

Output Parameters

<i>byall</i>	(global). Global size <i>n</i> , local size <i>n</i> . <i>byall</i> is exactly duplicated on all processes. It contains the same values as <i>bycol</i> , but it is replicated across all processes rather than being distributed.
--------------	---

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lared2d

*Redistributes an array assuming that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow*.*

Syntax

```
void pslared2d (MKL_INT *n , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , float *byrow ,
float *byall , float *work , MKL_INT *lwork );

void pdlared2d (MKL_INT *n , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , double
*byrow , double *byall , double *work , MKL_INT *lwork );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?lared2d` function redistributes a 1D array. It assumes that the input array *byrow* is distributed across columns and that all process rows contain the same copy of *byrow*. The output array *byall* will be identical on all processes and will contain the entire array.

Input Parameters

np = Number of local rows in *byrow*()

n (global)
The number of elements to be redistributed. $n \geq 0$.

ia, ja (global) *ia, ja* must be equal to 1.

desc (local) array of size *dlen_*. A 2D array descriptor, which describes *byrow*.

<i>byrow</i>	(local). Distributed block cyclic array of global size n and of local size np . <i>byrow</i> is distributed across the process columns. All process rows are assumed to contain the same value.
<i>work</i>	(local). size <i>lwork</i> . Used to hold the buffers sent from one process to another.
<i>lwork</i>	(local) The size of the <i>work</i> array. $lwork \geq \text{numroc}(n, \text{desc}[nb_], 0, 0, npcol)$.

Output Parameters

<i>byall</i>	(global). Global size n , local size n . <i>byall</i> is exactly duplicated on all processes. It contains the same values as <i>byrow</i> , but it is replicated across all processes rather than being distributed.
--------------	---

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
void pslarf (char *side , MKL_INT *m , MKL_INT *n , float *v , MKL_INT *iv , MKL_INT
*jv , MKL_INT *descv , MKL_INT *incv , float *tau , float *c , MKL_INT *ic , MKL_INT
*jc , MKL_INT *descc , float *work );

void pdlarf (char *side , MKL_INT *m , MKL_INT *n , double *v , MKL_INT *iv , MKL_INT
*jv , MKL_INT *descv , MKL_INT *incv , double *tau , double *c , MKL_INT *ic , MKL_INT
*jc , MKL_INT *descc , double *work );

void pclarf (char *side , MKL_INT *m , MKL_INT *n , MKL_Complex8 *v , MKL_INT *iv ,
MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex8 *tau , MKL_Complex8 *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work );

void pzlarf (char *side , MKL_INT *m , MKL_INT *n , MKL_Complex16 *v , MKL_INT *iv ,
MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex16 *tau , MKL_Complex16 *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description

The `p?larf` function applies a real/complex elementary reflector Q (or Q^T) to a real/complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau v v',$$

where τ is a real/complex scalar and v is a real/complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Input Parameters

<i>side</i>	(global). = 'L': form $Q * \text{sub}(C)$, = 'R': form $\text{sub}(C) * Q$, $Q = Q^T$.
<i>m</i>	(global) The number of rows in the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>v</i>	(local). Pointer into the local memory to an array of size $\text{lld}_v * \text{LOCc}(n_v)$, containing the local pieces of the global distributed matrix V representing the Householder transformation Q , $V(iv:iv+m-1, jv)$ if $side = 'L'$ and $incv = 1$, $V(iv, jv:jv+m-1)$ if $side = 'L'$ and $incv = m_v$, $V(iv:iv+n-1, jv)$ if $side = 'R'$ and $incv = 1$, $V(iv, jv:jv+n-1)$ if $side = 'R'$ and $incv = m_v$. The array v is the representation of Q . v is not used if $\tau = 0$.
<i>iv, jv</i>	(global) The row and column indices in the global matrix V indicating the first row and the first column of the matrix $\text{sub}(V)$, respectively.
<i>descv</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix V .
<i>incv</i>	(global) The global increment for the elements of V . Only two values of $incv$ are supported in this version, namely 1 and m_v . $incv$ must not be zero.
<i>tau</i>	(local). Array of size $\text{LOCc}(jv)$ if $incv = 1$, and $\text{LOCr}(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors. τ is tied to the distributed matrix V .
<i>c</i>	(local). Pointer into the local memory to an array of size $\text{lld}_c * \text{LOCc}(jc+n-1)$, containing the local pieces of $\text{sub}(C)$.
<i>ic, jc</i>	(global) The row and column indices in the global matrix C indicating the first row and the first column of the matrix $\text{sub}(C)$, respectively.
<i>desc</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix C .
<i>work</i>	(local).

Array of size *lwork*.

If *incv* = 1,

if *side* = 'L',

if *ivcol* = *iccol*,

$lwork \geq nqc0$

else

$lwork \geq mpc0 + \max(1, nqc0)$

end if

else if *side* = 'R',

$lwork \geq nqc0 + \max(\max(1, mpc0), \text{numroc}(\text{numroc}(n +$
 $\text{icoffc}, nb_v, 0, 0, npc0), nb_v, 0, 0, lcmq)))$

end if

else if *incv* = *m_v*,

if *side* = 'L',

$lwork \geq mpc0 + \max(\max(1, nqc0), \text{numroc}(\text{numroc}(m + iroffc, mb_v, 0, 0, nprow), mb_v, 0, 0, lcmq)))$

else if *side* = 'R',

if *ivrow* = *icrow*,

$lwork \geq mpc0$

else

$lwork \geq nqc0 + \max(1, mpc0)$

end if

end if

end if,

where *lcm* is the least common multiple of *nprow* and *npcol* and $lcm = \text{ilcm}(nprow, npc0)$, $lcmp = lcm/nprow$, $lcmq = lcm/npcol$,

$iroffc = \text{mod}(ic-1, mb_c)$, $icoffc = \text{mod}(jc-1, nb_c)$,

$icrow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow)$,

$iccol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, npc0)$,

$mpc0 = \text{numroc}(m + iroffc, mb_c, myrow, icrow, nprow)$,

$nqc0 = \text{numroc}(n + icoffc, nb_c, mycol, iccol, npc0)$,

ilcm, *indxg2p*, and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the function *blacs_gridinfo*.

Output Parameters

c

(local).

On exit, *sub(C)* is overwritten by the $Q * \text{sub}(C)$ if *side* = 'L',

or $\text{sub}(C) * Q$ if $\text{side} = 'R'$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```
void pslarfb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , float *v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , float
*t , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , float *work );

void pdlarfb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , double *v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv ,
double *t , double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , double *work );

void pclarfb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_Complex8 *v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv ,
MKL_Complex8 *t , MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc ,
MKL_Complex8 *work );

void pzlarfb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_Complex16 *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , MKL_Complex16 *t , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT
*descc , MKL_Complex16 *work );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?larfb` function applies a real/complex block reflector Q or its transpose Q^T /conjugate transpose Q^H to a real/complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Input Parameters

<i>side</i>	(global) if <i>side</i> = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the Left; if <i>side</i> = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the Right.
<i>trans</i>	(global) if <i>trans</i> = 'N': no transpose, apply Q ; for real flavors, if <i>trans</i> ='T': transpose, apply Q^T for complex flavors, if <i>trans</i> = 'C': conjugate transpose, apply Q^H ;
<i>direct</i>	(global) Indicates how Q is formed from a product of elementary reflectors. if <i>direct</i> = 'F': $Q = H(1) * H(2) * \dots * H(k)$ (Forward) if <i>direct</i> = 'B': $Q = H(k) * \dots * H(2) * H(1)$ (Backward)

<i>storev</i>	<p>(global)</p> <p>Indicates how the vectors that define the elementary reflectors are stored:</p> <p>if <i>storev</i> = 'C': Columnwise</p> <p>if <i>storev</i> = 'R': Rowwise.</p>
<i>m</i>	<p>(global)</p> <p>The number of rows in the distributed matrix sub(C). ($m \geq 0$).</p>
<i>n</i>	<p>(global)</p> <p>The number of columns in the distributed matrix sub(C). ($n \geq 0$).</p>
<i>k</i>	<p>(global)</p> <p>The order of the matrix T.</p>
<i>v</i>	<p>(local).</p> <p>Pointer into the local memory to an array of size</p> <p>$lld_v * LOCr(jv+k-1)$ if <i>storev</i> = 'C',</p> <p>$lld_v * LOCr(jv+m-1)$ if <i>storev</i> = 'R' and <i>side</i> = 'L',</p> <p>$lld_v * LOCr(jv+n-1)$ if <i>storev</i> = 'R' and <i>side</i> = 'R'.</p> <p>It contains the local pieces of the distributed vectors <i>V</i> representing the Householder transformation.</p> <p>if <i>storev</i> = 'C' and <i>side</i> = 'L', $lld_v \geq \max(1, LOCr(iv+m-1))$;</p> <p>if <i>storev</i> = 'C' and <i>side</i> = 'R', $lld_v \geq \max(1, LOCr(iv+n-1))$;</p> <p>if <i>storev</i> = 'R', $lld_v \geq LOCr(jv+k-1)$.</p>
<i>iv, jv</i>	<p>(global)</p> <p>The row and column indices in the global matrix <i>V</i> indicating the first row and the first column of the matrix sub(<i>V</i>), respectively.</p>
<i>descv</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>V</i>.</p>
<i>c</i>	<p>(local).</p> <p>Pointer into the local memory to an array of size $lld_c * LOCr(jc+n-1)$, containing the local pieces of sub(C).</p>
<i>ic, jc</i>	<p>(global) The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the matrix sub(C), respectively.</p>
<i>descc</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>C</i>.</p>
<i>work</i>	<p>(local).</p> <p>Workspace array of size <i>lwork</i>.</p> <p>If <i>storev</i> = 'C',</p> <p>if <i>side</i> = 'L',</p> <p>$lwork \geq (nqc0 + mpc0) * k$</p>

```

else if side = 'R',
    lwork ≥ ( nqc0 + max( npv0 + numroc( numroc( n +
        icoffc, nb_v, 0, 0, npcol ), nb_v, 0, 0, lcmq ),
        mpc0 ) ) * k
end if
else if storev = 'R' ,
    if side = 'L' ,
        lwork ≥ ( mpc0 + max( mqv0 + numroc( numroc( m +
            iroffc, mb_v, 0, 0, nprow ), mb_v, 0, 0, lcmp ),
            nqc0 ) ) * k
    else if side = 'R',
        lwork ≥ ( mpc0 + nqc0 ) * k
    end if
end if,
where
lcmq = lcm / npcol with lcm = iclm( nprow, npcol ),
iroffv = mod( iv-1, mb_v ), icoffv = mod( jv-1, nb_v ),
ivrow = indxg2p( iv, mb_v, myrow, rsrc_v, nprow ),
ivcol = indxg2p( jv, nb_v, mycol, csrc_v, npcol ),
MqV0 = numroc( m+icoffv, nb_v, mycol, ivcol, npcol ),
NpV0 = numroc( n+iroffv, mb_v, myrow, ivrow, nprow ),
iroffc = mod( ic-1, mb_c ), icoffc = mod( jc-1, nb_c ),
icrow = indxg2p( ic, mb_c, myrow, rsrc_c, nprow ),
iccol = indxg2p( jc, nb_c, mycol, csrc_c, npcol ),
MpC0 = numroc( m+iroffc, mb_c, myrow, icrow, nprow ),
NpC0 = numroc( n+icoffc, mb_c, myrow, icrow, nprow ),
NqC0 = numroc( n+icoffc, nb_c, mycol, iccol, npcol ),
iclm, indxg2p, and numroc are ScaLAPACK tool functions; myrow, mycol,
nprow, and npcol can be determined by calling the function
blacs_gridinfo.

```

Output Parameters

t (local).
 Array of size *mb_v* * *mb_vif* *storev* = 'R', and *nb_v* * *nb_vif* *storev* = 'C'. The triangular matrix *t* is the representation of the block reflector.

c (local).
 On exit, *sub(C)* is overwritten by the *Q***sub(C)*, or *Q'***sub(C)*, or *sub(C)***Q*, or *sub(C)***Q'*. *Q'* is transpose (conjugate transpose) of *Q*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larfc

Applies the conjugate transpose of an elementary reflector to a general matrix.

Syntax

```
void pclarfc (char *side , MKL_INT *m , MKL_INT *n , MKL_Complex8 *v , MKL_INT *iv ,
MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex8 *tau , MKL_Complex8 *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work );

void pzlarfc (char *side , MKL_INT *m , MKL_INT *n , MKL_Complex16 *v , MKL_INT *iv ,
MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex16 *tau , MKL_Complex16 *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description

The `p?larfc` function applies a complex elementary reflector Q^H to a complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau * v * v',$$

where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Input Parameters

<code>side</code>	(global) if <code>side = 'L'</code> : form $Q^H * \text{sub}(C)$; if <code>side = 'R'</code> : form $\text{sub}(C) * Q^H$.
<code>m</code>	(global) The number of rows in the distributed matrix $\text{sub}(C)$. ($m \geq 0$).
<code>n</code>	(global) The number of columns in the distributed matrix $\text{sub}(C)$. ($n \geq 0$).
<code>v</code>	(local). Pointer into the local memory to an array of size $lld_v * LOCC(n_v)$, containing the local pieces of the global distributed matrix V representing the Householder transformation Q , $V(iv:iv+m-1, jv)$ if <code>side = 'L'</code> and <code>incv = 1</code> , $V(iv, jv:jv+m-1)$ if <code>side = 'L'</code> and <code>incv = m_v</code> , $V(iv:iv+n-1, jv)$ if <code>side = 'R'</code> and <code>incv = 1</code> , $V(iv, jv:jv+n-1)$ if <code>side = 'R'</code> and <code>incv = m_v</code> . The array <code>v</code> is the representation of Q . <code>v</code> is not used if $\tau = 0$.

<i>iv, jv</i>	(global) The row and column indices in the global matrix <i>V</i> indicating the first row and the first column of the matrix sub(<i>V</i>), respectively.
<i>descv</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>V</i> .
<i>incv</i>	(global) The global increment for the elements of <i>v</i> . Only two values of <i>incv</i> are supported in this version, namely 1 and <i>m_v</i> . <i>incv</i> must not be zero.
<i>tau</i>	(local) Array of size <i>LOCc(jv)</i> if <i>incv</i> = 1, and <i>LOCr(iv)</i> otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix <i>V</i> .
<i>c</i>	(local). Pointer into the local memory to an array of size <i>lld_c</i> * <i>LOCc(jc+n-1)</i> , containing the local pieces of sub(<i>C</i>).
<i>ic, jc</i>	(global) The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the matrix sub(<i>C</i>), respectively.
<i>descc</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local). Workspace array of size <i>lwork</i> . If <i>incv</i> = 1, if <i>side</i> = 'L', if <i>ivcol</i> = <i>iccol</i> , <i>lwork</i> ≥ <i>nqc0</i> else <i>lwork</i> ≥ <i>mpc0</i> + max(1, <i>nqc0</i>) end if else if <i>side</i> = 'R', <i>lwork</i> ≥ <i>nqc0</i> + max(max(1, <i>mpc0</i>), numroc(numroc(<i>n+icoffc,nb_v,0,0,npcol</i>), <i>nb_v,0,0,lcmq</i>)) end if else if <i>incv</i> = <i>m_v</i> , if <i>side</i> = 'L', <i>lwork</i> ≥ <i>mpc0</i> + max(max(1, <i>nqc0</i>), numroc(numroc(<i>m+iroffc,mb_v,0,0,nprow</i>), <i>mb_v,0,0,lcmp</i>))

```

else if side = 'R' ,
    if ivrow = icrow,
        lwork ≥ mpc0
    else
        lwork ≥ nqc0 + max( 1, mpc0 )
    end if
end if
end if,
where lcm is the least common multiple of nprow and npcol and lcm =
ilcm(nprow, npcol),
 $lcmp = lcm/nprow$ ,  $lcmq = lcm/npcol$ ,
 $iroffc = \text{mod}(ic-1, mb\_c)$ ,  $icoffc = \text{mod}(jc-1, nb\_c)$ ,
 $icrow = \text{indxg2p}(ic, mb\_c, myrow, rsrc\_c, nprow)$ ,
 $iccol = \text{indxg2p}(jc, nb\_c, mycol, csrc\_c, npc0)$ ,
 $mpc0 = \text{numroc}(m+iroffc, mb\_c, myrow, icrow, nprow)$ ,
 $nqc0 = \text{numroc}(n+icoffc, nb\_c, mycol, iccol, npc0)$ ,
ilcm,  $\text{indxg2p}$ , and  $\text{numroc}$  are ScaLAPACK tool functions; myrow, mycol,
nprow, and npcol can be determined by calling the function
blacs_gridinfo.

```

Output Parameters

c (local).
On exit, *sub(C)* is overwritten by the $Q^H \cdot \text{sub}(C)$ if *side* = 'L', or *sub(C)* * Q^H if *side* = 'R'.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```

void pslarfg (MKL_INT *n , float *alpha , MKL_INT *iax , MKL_INT *jax , float *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , float *tau );

void pdlarfg (MKL_INT *n , double *alpha , MKL_INT *iax , MKL_INT *jax , double *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *tau );

void pclarfg (MKL_INT *n , MKL_Complex8 *alpha , MKL_INT *iax , MKL_INT *jax ,
MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx ,
MKL_Complex8 *tau );

void pzlarfg (MKL_INT *n , MKL_Complex16 *alpha , MKL_INT *iax , MKL_INT *jax ,
MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx ,
MKL_Complex16 *tau );

```

Include Files

- `mkl_scalapack.h`

Description

The `p?larfg` function generates a real/complex elementary reflector H of order n , such that

$$H * \text{sub}(X) = H * \begin{pmatrix} x(iax, jax) \\ x \end{pmatrix} = \begin{pmatrix} \alpha \\ 0 \end{pmatrix}, \quad H^* H = I,$$

where α is a scalar (a real scalar - for complex flavors), and $\text{sub}(X)$ is an $(n-1)$ -element real/complex distributed vector $X(ix:ix+n-2, jx)$ if $incx = 1$ and $X(ix, jx:jx+n-2)$ if $incx = m_x$. H is represented in the form

$$H = I - \tau * \begin{pmatrix} 1 \\ v \end{pmatrix} * (1 \ v')$$

where τ is a real/complex scalar and v is a real/complex $(n-1)$ -element vector. Note that H is not Hermitian.

If the elements of $\text{sub}(X)$ are all zero (and $X(iax, jax)$ is real for complex flavors), then $\tau = 0$ and H is taken to be the unit matrix.

Otherwise $1 \leq \text{real}(\tau) \leq 2$ and $\text{abs}(\tau-1) \leq 1$.

Input Parameters

n	(global) The global order of the elementary reflector. $n \geq 0$.
iax, jax	(global) The global row and column indices of $X(iax, jax)$ in the global matrix X .
x	(local). Pointer into the local memory to an array of size $lld_x * LOCC(n_x)$. This array contains the local pieces of the distributed vector $\text{sub}(X)$. Before entry, the incremented array $\text{sub}(X)$ must contain vector x .
ix, jx	(global) The row and column indices in the global matrix X indicating the first row and the first column of $\text{sub}(X)$, respectively.
$descx$	(global and local) Array of size $dlen_*$. The array descriptor for the distributed matrix X .

incx (global)
The global increment for the elements of *x*. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

alpha (local)
On exit, *alpha* is computed in the process scope having the vector sub(*X*).

x (local).
On exit, it is overwritten with the vector *v*.

tau (local).
Array of size *LOCc(jx)* if *incx* = 1, and *LOCr(ix)* otherwise. This array contains the Householder scalars related to the Householder vectors.
tau is tied to the distributed matrix *X*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larft

*Forms the triangular vector T of a block reflector $H=I-V^*T^*V^H$.*

Syntax

```
void pslarft (char *direct , char *storev , MKL_INT *n , MKL_INT *k , float *v , MKL_INT
*iv , MKL_INT *jv , MKL_INT *descv , float *tau , float *t , float *work );

void pdlarft (char *direct , char *storev , MKL_INT *n , MKL_INT *k , double *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , double *tau , double *t , double *work );

void pclarft (char *direct , char *storev , MKL_INT *n , MKL_INT *k , MKL_Complex8 *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_Complex8 *tau , MKL_Complex8 *t ,
MKL_Complex8 *work );

void pzlarft (char *direct , char *storev , MKL_INT *n , MKL_INT *k , MKL_Complex16
*v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_Complex16 *tau , MKL_Complex16
*t , MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description

The *p?larft* function forms the triangular factor *T* of a real/complex block reflector *H* of order *n*, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', $H = H(1) * H(2) \dots * H(k)$, and *T* is upper triangular;

If *direct* = 'B', $H = H(k) * \dots * H(2) * H(1)$, and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th column of the distributed matrix *V*, and

$$H = I - V * T * V'$$

If $storev = 'R'$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th row of the distributed matrix V , and

$$H = I - V' * T * V.$$

Input Parameters

<i>direct</i>	(global) Specifies the order in which the elementary reflectors are multiplied to form the block reflector: if $direct = 'F'$: $H = H(1) * H(2) * \dots * H(k)$ (forward) if $direct = 'B'$: $H = H(k) * \dots * H(2) * H(1)$ (backward).
<i>storev</i>	(global) Specifies how the vectors that define the elementary reflectors are stored (See <i>Application Notes</i> below): if $storev = 'C'$: columnwise; if $storev = 'R'$: rowwise.
<i>n</i>	(global) The order of the block reflector H . $n \geq 0$.
<i>k</i>	(global) The order of the triangular factor T , is equal to the number of elementary reflectors. $1 \leq k \leq mb_v (= nb_v)$.
<i>v</i>	Pointer into the local memory to an array of local size $LOCr(iv+n-1) * LOCc(jv+k-1)$ if $storev = 'C'$, and $LOCr(iv+k-1) * LOCc(jv+n-1)$ if $storev = 'R'$. The distributed matrix V contains the Householder vectors. (See <i>Application Notes</i> below).
<i>iv, jv</i>	(global) The row and column indices in the global matrix V indicating the first row and the first column of the matrix $sub(V)$, respectively.
<i>descv</i>	(local) array of size $dlen_$. The array descriptor for the distributed matrix V .
<i>tau</i>	(local) Array of size $LOCr(iv+k-1)$ if $incv = m_v$, and $LOCc(jv+k-1)$ otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix V .
<i>work</i>	(local). Workspace array of size $k*(k-1)/2$.

Output Parameters

v

t

(local)

Array of size $nb_v * nb_v$ if $storev = 'C'$, and $mb_v * mb_v$ otherwise. It contains the k -by- k triangular factor of the block reflector associated with v . If $direct = 'F'$, t is upper triangular; if $direct = 'B'$, t is lower triangular.

Application Notes

The shape of the matrix V and the storage of the vectors that define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

$direct = 'F'$ and $storev = 'C'$:

$$V(iv : iv + n - 1, jv : jv + k - 1) = \begin{bmatrix} 1 & & & & \\ v1 & 1 & & & \\ v1 & v2 & 1 & & \\ v1 & v2 & v3 & & \\ v1 & v2 & v3 & & \end{bmatrix}$$

$direct = 'F'$ and $storev = 'R'$

$$V(iv : iv + k - 1, jv : jv + n - 1) = \begin{bmatrix} 1 & v1 & v1 & v1 & v1 \\ & 1 & v2 & v2 & v2 \\ & & 1 & v3 & v3 \end{bmatrix}$$

$direct = 'B'$ and $storev = 'C'$

$$V(iv : iv + n - 1, jv : jv + k - 1) = \begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ 1 & v2 & v3 \\ & 1 & v3 \\ & & 1 \end{bmatrix}$$

$direct = 'B'$ and $storev = 'R'$

$$V(iv : iv + k - 1, jv : jv + n - 1) = \begin{bmatrix} v1 & v1 & 1 \\ v2 & v2 & v2 & 1 \\ v3 & v3 & v3 & v3 & 1 \end{bmatrix}$$

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larz

Applies an elementary reflector as returned by p?tzzrf to a general matrix.

Syntax

```
void pslarz (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , float *v , MKL_INT
*iv , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , float *tau , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , float *work );
```

```
void pdlarz (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , double *v , MKL_INT
*iv , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work );
```

```

void pclarz (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex8 *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , MKL_Complex8 *work );

void pzlarz (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex16 *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , MKL_Complex16 *work );

```

Include Files

- mkl_scalapack.h

Description

The `p?larz` function applies a real/complex elementary reflector Q (or Q^T) to a real/complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau v v',$$

where τ is a real/complex scalar and v is a real/complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Q is a product of k elementary reflectors as returned by `p?tzzrf`.

Input Parameters

<code>side</code>	(global) if <code>side = 'L'</code> : form $Q * \text{sub}(C)$, if <code>side = 'R'</code> : form $\text{sub}(C) * Q$, $Q = Q^T$ (for real flavors).
<code>m</code>	(global) The number of rows in the distributed matrix $\text{sub}(C)$. ($m \geq 0$).
<code>n</code>	(global) The number of columns in the distributed matrix $\text{sub}(C)$. ($n \geq 0$).
<code>l</code>	(global) The columns of the distributed matrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If <code>side = 'L'</code> , $m \geq l \geq 0$, if <code>side = 'R'</code> , $n \geq l \geq 0$.
<code>v</code>	(local). Pointer into the local memory to an array of size $lld_v * LOCC(n_v)$ containing the local pieces of the global distributed matrix V representing the Householder transformation Q , $V(iv:iv+l-1, jv)$ if <code>side = 'L'</code> and <code>incv = 1</code> , $V(iv, jv:jv+l-1)$ if <code>side = 'L'</code> and <code>incv = m_v</code> , $V(iv:iv+l-1, jv)$ if <code>side = 'R'</code> and <code>incv = 1</code> , $V(iv, jv:jv+l-1)$ if <code>side = 'R'</code> and <code>incv = m_v</code> . The vector v in the representation of Q . v is not used if $\tau = 0$.

<i>iv, jv</i>	(global) The row and column indices in the global distributed matrix <i>V</i> indicating the first row and the first column of the matrix sub(<i>V</i>), respectively.
<i>descv</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>V</i> .
<i>incv</i>	(global) The global increment for the elements of <i>V</i> . Only two values of <i>incv</i> are supported in this version, namely 1 and <i>m_v</i> . <i>incv</i> must not be zero.
<i>tau</i>	(local) Array of size <i>LOCc(jv)</i> if <i>incv</i> = 1, and <i>LOCr(iv)</i> otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix <i>V</i> .
<i>c</i>	(local). Pointer into the local memory to an array of size <i>lld_c</i> * <i>LOCc(jc+n-1)</i> , containing the local pieces of sub(<i>C</i>).
<i>ic, jc</i>	(global) The row and column indices in the global matrix <i>C</i> indicating the first row and the first column of the matrix sub(<i>C</i>), respectively.
<i>descC</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local). Array of size <i>lwork</i> If <i>incv</i> = 1, if <i>side</i> = 'L' , if <i>ivcol</i> = <i>iccol</i> , <i>lwork</i> ≥ <i>NqC0</i> else <i>lwork</i> ≥ <i>MpC0</i> + max(1, <i>NqC0</i>) end if else if <i>side</i> = 'R' , <i>lwork</i> ≥ <i>NqC0</i> + max(max(1, <i>MpC0</i>), numroc(numroc(<i>n</i> + <i>icoffc</i> , <i>nb_v</i> , 0, 0, <i>npcol</i>), <i>nb_v</i> , 0, 0, <i>lcmq</i>)) end if else if <i>incv</i> = <i>m_v</i> , if <i>side</i> = 'L' , <i>lwork</i> ≥ <i>MpC0</i> + max(max(1, <i>NqC0</i>), numroc(numroc(<i>m</i> + <i>irowfc</i> , <i>mb_v</i> , 0, 0, <i>nprow</i>), <i>mb_v</i> , 0, 0, <i>lcmp</i>)) else if <i>side</i> = 'R' ,

```

        if ivrow = icrow,
            lwork ≥ MpC0
        else
            lwork ≥ NqC0 + max(1, MpC0)
        end if
    end if
end if.

Here lcm is the least common multiple of nprow and npc0l and
lcm = ilcm( nprow, npc0l ), lcmp = lcm / nprow,
lcmq = lcm / npc0l,
iroffc = mod( ic-1, mb_c ), icoffc = mod( jc-1, nb_c ),
icrow = indxg2p( ic, mb_c, myrow, rsrc_c, nprow ),
iccol = indxg2p( jc, nb_c, mycol, csrc_c, npc0l ),
mpc0 = numroc( m+iroffc, mb_c, myrow, icrow, nprow ),
nqc0 = numroc( n+icoffc, nb_c, mycol, iccol, npc0l ),

ilcm, indxg2p, and numroc are ScaLAPACK tool functions; myrow, mycol,
nprow, and npc0l can be determined by calling the function
blacs_gridinfo.

```

Output Parameters

C

(local).

On exit, sub(C) is overwritten by the $Q \cdot \text{sub}(C)$ if *side* = 'L', or $\text{sub}(C) \cdot Q$ if *side* = 'R'.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larzb

Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzzrf to a general matrix.

Syntax

```
void pslarzb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_INT *l , float *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , float *t , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , float
*work );
```

```
void pdlarzb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_INT *l , double *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , double *t , double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , double
*work );
```

```

void pclarzb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_INT *l , MKL_Complex8 *v , MKL_INT *iv , MKL_INT *jv ,
MKL_INT *descv , MKL_Complex8 *t , MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc , MKL_Complex8 *work );

void pzlarzb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_INT *l , MKL_Complex16 *v , MKL_INT *iv , MKL_INT *jv ,
MKL_INT *descv , MKL_Complex16 *t , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc , MKL_Complex16 *work );

```

Include Files

- mkl_scalapack.h

Description

The `p?larzbb` function applies a real/complex block reflector Q or its transpose Q^T (conjugate transpose Q^H for complex flavors) to a real/complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Q is a product of k elementary reflectors as returned by `p?tzrzf`.

Currently, only `storev = 'R'` and `direct = 'B'` are supported.

Input Parameters

<code>side</code>	(global) if <code>side = 'L'</code> : apply Q or Q^T (Q^H for complex flavors) from the Left; if <code>side = 'R'</code> : apply Q or Q^T (Q^H for complex flavors) from the Right.
<code>trans</code>	(global) if <code>trans = 'N'</code> : No transpose, apply Q ; If <code>trans='T'</code> : Transpose, apply Q^T (real flavors); If <code>trans='C'</code> : Conjugate transpose, apply Q^H (complex flavors).
<code>direct</code>	(global) Indicates how H is formed from a product of elementary reflectors. if <code>direct = 'F'</code> : $H = H(1)*H(2)*\dots*H(k)$ - forward (not supported) ; if <code>direct = 'B'</code> : $H = H(k)*\dots*H(2)*H(1)$ - backward.
<code>storev</code>	(global) Indicates how the vectors that define the elementary reflectors are stored: if <code>storev = 'C'</code> : columnwise (not supported). if <code>storev = 'R'</code> : rowwise.
<code>m</code>	(global) The number of rows in the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).
<code>n</code>	(global) The number of columns in the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).
<code>k</code>	(global)

	<p>The order of the matrix T. (= the number of elementary reflectors whose product defines the block reflector).</p>
l	<p>(global)</p> <p>The columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors.</p> <p>If $side = 'L', m \geq l \geq 0$,</p> <p>if $side = 'R', n \geq l \geq 0$.</p>
v	<p>(local).</p> <p>Pointer into the local memory to an array of size $lld_v * LOCc(j_v+m-1)$ if $side = 'L', lld_v * LOCc(j_v+m-1)$ if $side = 'R'$.</p> <p>It contains the local pieces of the distributed vectors V representing the Householder transformation as returned by <code>p?tzrzf</code>.</p> <p>$lld_v \geq LOCr(iv+k-1)$.</p>
iv, jv	<p>(global)</p> <p>The row and column indices in the global matrix V indicating the first row and the first column of the submatrix $\text{sub}(V)$, respectively.</p>
$descv$	<p>(global and local) array of size $dlen_$. The array descriptor for the distributed matrix V.</p>
t	<p>(local)</p> <p>Array of size $mb_v * mb_v$.</p> <p>The lower triangular matrix T in the representation of the block reflector.</p>
c	<p>(local).</p> <p>Pointer into the local memory to an array of size $lld_c * LOCc(j_c+n-1)$.</p> <p>On entry, the m-by-n distributed matrix $\text{sub}(C)$.</p>
ic, jc	<p>(global)</p> <p>The row and column indices in the global matrix C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.</p>
$descc$	<p>(global and local) array of size $dlen_$. The array descriptor for the distributed matrix C.</p>
$work$	<p>(local).</p>

Array of size *lwork*.

```

If storev = 'C' ,
    if side = 'L' ,
        lwork ≥ (nqc0 + mpc0) * k
    else if side = 'R' ,
        lwork ≥ (nqc0 + max(npv0 + numroc(numroc(n+icoffc, nb_v, 0, 0,
            npcol),
                nb_v, 0, 0, lcmq), mpc0)) * k
    end if
else if storev = 'R' ,
    if side = 'L' ,
        lwork ≥ (mpc0 + max(mqv0 + numroc(numroc(m+iroffc, mb_v, 0, 0,
            nprow),
                mb_v, 0, 0, lcmp), nqc0)) * k
    else if side = 'R' ,
        lwork ≥ (mpc0 + nqc0) * k
    end if
end if.

```

Here $lcmq = lcm/npcol$ with $lcm = iclm(nprow, npcol)$,
 $iroffv = \text{mod}(iv-1, mb_v)$, $icoffv = \text{mod}(jv-1, nb_v)$,
 $ivrow = \text{indxg2p}(iv, mb_v, myrow, rsrc_v, nprow)$,
 $ivcol = \text{indxg2p}(jv, nb_v, mycol, csrc_v, npcol)$,
 $mqv0 = \text{numroc}(m+icoffv, nb_v, mycol, ivcol, npcol)$,
 $npv0 = \text{numroc}(n+iroffv, mb_v, myrow, ivrow, nprow)$,
 $iroffc = \text{mod}(ic-1, mb_c)$, $icoffc = \text{mod}(jc-1, nb_c)$,
 $icrow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow)$,
 $iccol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, npcol)$,
 $mpc0 = \text{numroc}(m+iroffc, mb_c, myrow, icrow, nprow)$,
 $npc0 = \text{numroc}(n+icoffc, mb_c, myrow, icrow, nprow)$,
 $nqc0 = \text{numroc}(n+icoffc, nb_c, mycol, iccol, npcol)$,
 $iclm$, indxg2p , and numroc are ScaLAPACK tool functions; $myrow$, $mycol$,
 $nprow$, and $npcol$ can be determined by calling the function
`blacs_gridinfo`.

Output Parameters

c

(local).

On exit, $\text{sub}(C)$ is overwritten by the $Q^H \text{sub}(C)$, or $Q'^H \text{sub}(C)$, or $\text{sub}(C) * Q$, or $\text{sub}(C) * Q'$, where Q' is the transpose (conjugate transpose) of Q .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larzc

Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzzrf to a general matrix.

Syntax

```
void pclarzc (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex8 *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , MKL_Complex8 *work );

void pzlarzc (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex16 *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description

The `p?larzc` function applies a complex elementary reflector Q^H to a complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau * v * v',$$

where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Q is a product of k elementary reflectors as returned by `p?tzzrf`.

Input Parameters

<code>side</code>	(global) if <code>side = 'L'</code> : form $Q^H * \text{sub}(C)$; if <code>side = 'R'</code> : form $\text{sub}(C) * Q^H$.
<code>m</code>	(global) The number of rows in the distributed matrix $\text{sub}(C)$. ($m \geq 0$).
<code>n</code>	(global) The number of columns in the distributed matrix $\text{sub}(C)$. ($n \geq 0$).
<code>l</code>	(global) The columns of the distributed matrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If <code>side = 'L'</code> , $m \geq l \geq 0$,

	if $side = 'R', n \geq l \geq 0$.
v	<p>(local).</p> <p>Pointer into the local memory to an array of size $lld_v * LOCc(n_v)$ containing the local pieces of the global distributed matrix V representing the Householder transformation Q,</p> <p>$V(iv:iv+l-1, jv)$ if $side = 'L'$ and $incv = 1$, $V(iv, jv:jv+l-1)$ if $side = 'L'$ and $incv = m_v$, $V(iv:iv+l-1, jv)$ if $side = 'R'$ and $incv = 1$, $V(iv, jv:jv+l-1)$ if $side = 'R'$ and $incv = m_v$.</p> <p>The vector v in the representation of Q. v is not used if $tau = 0$.</p>
iv, jv	<p>(global)</p> <p>The row and column indices in the global matrix V indicating the first row and the first column of the matrix $sub(V)$, respectively.</p>
$descv$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix V .
$incv$	<p>(global)</p> <p>The global increment for the elements of V. Only two values of $incv$ are supported in this version, namely 1 and m_v.</p> <p>$incv$ must not be zero.</p>
tau	<p>(local)</p> <p>Array of size $LOCc(jv)$ if $incv = 1$, and $LOCr(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors.</p> <p>tau is tied to the distributed matrix V.</p>
c	<p>(local).</p> <p>Pointer into the local memory to an array of size $lld_c * LOCc(jc+n-1)$, containing the local pieces of $sub(C)$.</p>
ic, jc	<p>(global)</p> <p>The row and column indices in the global matrix C indicating the first row and the first column of the matrix $sub(C)$, respectively.</p>
$descC$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix C .
$work$	<p>(local).</p> <pre> If incv = 1, if side = 'L' , if ivcol = iccol, lwork ≥ nqc0 else lwork ≥ mpc0 + max(1, nqc0) end if else if side = 'R' , lwork ≥ nqc0 + max(max(1, mpc0), numroc(numroc(n+icoffc, nb_v, </pre>

```

0, 0, npcol),
    nb_v, 0, 0, lcmq)) end if
else if incv = m_v,
    if side = 'L' ,
        lwork ≥ mpc0 + max(max(1, nqc0), numroc(numroc(m+iroffc, mb_v,
0, 0, nprow),
            mb_v, 0, 0, lcmp))
    else if side = 'R',
        if ivrow = icrow,
            lwork ≥ mpc0
        else
            lwork ≥ nqc0 + max(1, mpc0)
        end if
    end if
end if
end if

```

Here *lcm* is the least common multiple of *nprow* and *npcol*;

```

lcm = ilcm(nprow, npcol), lcmp = lcm/nprow, lcmq= lcm/npcol,
iroffc = mod(ic-1, mb_c), icoffc= mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrc_c, npcol),
mpc0 = numroc(m+iroffc, mb_c, myrow, icrow, nprow),
nqc0 = numroc(n+icoffc, nb_c, mycol, iccol, npcol),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions;

```

myrow, *mycol*, *nprow*, and *npcol* can be determined by calling the function `blacs_gridinfo`.

Output Parameters

c

(local).

On exit, `sub(C)` is overwritten by the $Q^H \cdot \text{sub}(C)$ if *side* = 'L', or $\text{sub}(C) \cdot Q^H$ if *side* = 'R'.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?larzt

*Forms the triangular factor T of a block reflector H=I-V*T*V^H as returned by p?tzzrf.*

Syntax

```

void pslarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , float *v , MKL_INT
*iv , MKL_INT *jv , MKL_INT *descv , float *tau , float *t , float *work );

void pdlarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , double *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , double *tau , double *t , double *work );

void pclarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , MKL_Complex8 *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_Complex8 *tau , MKL_Complex8 *t ,
MKL_Complex8 *work );

```

```
void pzlarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , MKL_Complex16
*v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_Complex16 *tau , MKL_Complex16
*t , MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description

The `pzlarzt` function forms the triangular factor T of a real/complex block reflector H of order greater than n , which is defined as a product of k elementary reflectors as returned by `p?tzrzf`.

If `direct = 'F'`, $H = H(1)*H(2)*\dots*H(k)$, and T is upper triangular;

If `direct = 'B'`, $H = H(k)*\dots*H(2)*H(1)$, and T is lower triangular.

If `storev = 'C'`, the vector which defines the elementary reflector $H(i)$, is stored in the i -th column of the array v , and

$$H = I - v * t * v^H.$$

If `storev = 'R'`, the vector, which defines the elementary reflector $H(i)$, is stored in the i -th row of the array v , and

$$H = I - v^H * t * v.$$

Currently, only `storev = 'R'` and `direct = 'B'` are supported.

Input Parameters

<code>direct</code>	(global) Specifies the order in which the elementary reflectors are multiplied to form the block reflector: if <code>direct = 'F'</code> : $H = H(1)*H(2)*\dots*H(k)$ (Forward, not supported) if <code>direct = 'B'</code> : $H = H(k)*\dots*H(2)*H(1)$ (Backward).
<code>storev</code>	(global) Specifies how the vectors which defines the elementary reflectors are stored: if <code>storev = 'C'</code> : columnwise (not supported); if <code>storev = 'R'</code> : rowwise.
<code>n</code>	(global) The order of the block reflector H . $n \geq 0$.
<code>k</code>	(global) The order of the triangular factor T (= the number of elementary reflectors). $1 \leq k \leq mb_v (= nb_v)$.
<code>v</code>	Pointer into the local memory to an array of local size $LOCr(iv+k-1) * LOCc(jv+n-1)$. The distributed matrix V contains the Householder vectors. See <i>Application Notes</i> below.

<i>iv, jv</i>	(global) The row and column indices in the global matrix <i>V</i> indicating the first row and the first column of the matrix sub(<i>V</i>), respectively.
<i>descv</i>	(local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>V</i> .
<i>tau</i>	(local) Array of size <i>LOCr(iv+k-1)</i> if <i>incv = m_v</i> , and <i>LOCc(jv+k-1)</i> otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix <i>V</i> .
<i>work</i>	(local). Workspace array of size $(k * (k-1) / 2)$.

Output Parameters

<i>v</i>	
<i>t</i>	(local) Array of size <i>mb_v</i> * <i>mb_v</i> . It contains the <i>k</i> -by- <i>k</i> triangular factor of the block reflector associated with <i>v</i> . <i>t</i> is lower triangular.

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct='F' and storev='C'

$$\begin{array}{c}
 \begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix} \\
 v = \begin{array}{ccccc}
 & . & . & . & \\
 & . & . & . & \\
 1 & . & . & . & \\
 & 1 & . & . & \\
 & & 1 & . & \\
 & & & 1 & .
 \end{array}
 \end{array}$$

direct='F' and storev='R':

$$\begin{bmatrix} \overbrace{v1 \ v1 \ v1 \ v1 \ v1}^v & . & . & . & . & 1 \\ v2 \ v2 \ v2 \ v2 \ v2 & . & . & . & 1 \\ v3 \ v3 \ v3 \ v3 \ v3 & . & . & 1 \end{bmatrix}$$

direct='B' and storev='C':

$$v = \begin{bmatrix} 1 \\ . & 1 \\ . & . & 1 \\ . & . & . \\ . & . & . \end{bmatrix} \begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}$$

direct='B' and storev='R':

$$\begin{bmatrix} 1 & . & . & . & . & \overbrace{v1 \ v1 \ v1 \ v1 \ v1}^v \\ . & 1 & . & . & . & v2 \ v2 \ v2 \ v2 \ v2 \\ . & . & 1 & . & . & v3 \ v3 \ v3 \ v3 \ v3 \end{bmatrix}$$

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lascl

Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from} .

Syntax

```
void pslascl (char *type , float *cfrom , float *cto , MKL_INT *m , MKL_INT *n , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );

void pdlascl (char *type , double *cfrom , double *cto , MKL_INT *m , MKL_INT *n ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );

void pclascl (char *type , float *cfrom , float *cto , MKL_INT *m , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );

void pzlascl (char *type , double *cfrom , double *cto , MKL_INT *m , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?lascl` function multiplies the m -by- n real/complex distributed matrix $\text{sub}(A)$ denoting $A(ia:ia+m-1, ja:ja+n-1)$ by the real/complex scalar $cto/cfrom$. This is done without over/underflow as long as the final result $cto * A(i,j) / cfrom$ does not over/underflow. *type* specifies that $\text{sub}(A)$ may be full, upper triangular, lower triangular or upper Hessenberg.

Input Parameters

<i>type</i>	(global) <i>type</i> indicates the storage type of the input distributed matrix. if <i>type</i> = 'G': $\text{sub}(A)$ is a full matrix, if <i>type</i> = 'L': $\text{sub}(A)$ is a lower triangular matrix, if <i>type</i> = 'U': $\text{sub}(A)$ is an upper triangular matrix, if <i>type</i> = 'H': $\text{sub}(A)$ is an upper Hessenberg matrix.
<i>cfrom, cto</i>	(global) The distributed matrix $\text{sub}(A)$ is multiplied by $cto/cfrom$. $A(i,j)$ is computed without over/underflow if the final result $cto * A(i,j) / cfrom$ can be represented without over/underflow. <i>cfrom</i> must be nonzero.
<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local input/local output) Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) The column and row indices in the global matrix A indicating the first row and column of the matrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local)

Array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

<i>a</i>	(local). On exit, this array contains the local pieces of the distributed matrix multiplied by <i>cto/cfrom</i> .
<i>info</i>	(local) if <i>info</i> = 0: the execution is successful. if <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> -1, had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lase2

Initializes an m-by-n distributed matrix.

Syntax

```
void pslase2 (const char* uplo, const MKL_INT* m, const MKL_INT* n, const float* alpha,
const float* beta, float* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT*
desca);

void pdlase2 (const char* uplo, const MKL_INT* m, const MKL_INT* n, const double*
alpha, const double* beta, double* a, const MKL_INT* ia, const MKL_INT* ja, const
MKL_INT* desca);

void pclase2 (const char* uplo, const MKL_INT* m, const MKL_INT* n, const MKL_Complex8*
alpha, const MKL_Complex8* beta, MKL_Complex8* a, const MKL_INT* ia, const MKL_INT* ja,
const MKL_INT* desca);

void pzase2 (const char* uplo, const MKL_INT* m, const MKL_INT* n, const
MKL_Complex16* alpha, const MKL_Complex16* beta, MKL_Complex16* a, const MKL_INT* ia,
const MKL_INT* ja, const MKL_INT* desca);
```

Include Files

- mkl_scalapack.h

Description

p?lase2 initializes an *m*-by-*n* distributed matrix sub(*A*) denoting *A*(*ia:ia+m-1,ja:ja+n-1*) to *beta* on the diagonal and *alpha* on the off-diagonals. p?lase2 requires that only the dimension of the matrix operand is distributed.

Input Parameters

<i>uplo</i>	(global) Specifies the part of the distributed matrix sub(<i>A</i>) to be set: = 'U': Upper triangular part is set; the strictly lower triangular part of sub(<i>A</i>) is not changed;
-------------	---

= 'L': Lower triangular part is set; the strictly upper triangular part of $\text{sub}(A)$ is not changed;

Otherwise: All of the matrix $\text{sub}(A)$ is set.

m	(global) The number of rows to be operated on i.e the number of rows of the distributed submatrix $\text{sub}(A)$. $m \geq 0$.
n	(global) The number of columns to be operated on i.e the number of columns of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.
α	(global) The constant to which the off-diagonal elements are to be set.
β	(global) The constant to which the diagonal elements are to be set.
ia	(global) The row index in the global array a indicating the first row of $\text{sub}(A)$.
ja	(global) The column index in the global array a indicating the first column of $\text{sub}(A)$.
$desca$	(global and local) Array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

a	(local) Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$ to be set. On exit, the leading m -by- n submatrix $\text{sub}(A)$ is set as follows: if $uplo = 'U'$, $A(ia+i-1, ja+j-1) = \alpha$, $1 \leq i \leq j-1$, $1 \leq j \leq n$, if $uplo = 'L'$, $A(ia+i-1, ja+j-1) = \alpha$, $j+1 \leq i \leq m$, $1 \leq j \leq n$, otherwise, $A(ia+i-1, ja+j-1) = \alpha$, $1 \leq i \leq m$, $1 \leq j \leq n$, $ia+i \neq ja+j$, and, for all $uplo$, $A(ia+i-1, ja+i-1) = \beta$, $1 \leq i \leq \min(m, n)$.
-----	---

p?laset

Initializes the offdiagonal elements of a matrix to α and the diagonal elements to β .

Syntax

```
void pslaset (char *uplo , MKL_INT *m , MKL_INT *n , float *alpha , float *beta , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );

void pdlaset (char *uplo , MKL_INT *m , MKL_INT *n , double *alpha , double *beta ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );

void pclaset (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex8 *alpha , MKL_Complex8
*beta , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );

void pzaset (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex16 *alpha ,
MKL_Complex16 *beta , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
```

Include Files

- mkl_scalapack.h

Description

The `p?laset` function initializes an m -by- n distributed matrix $\text{sub}(A)$ denoting $A(ia:ia+m-1, ja:ja+n-1)$ to β on the diagonal and α on the offdiagonals.

Input Parameters

<i>uplo</i>	(global) Specifies the part of the distributed matrix $\text{sub}(A)$ to be set: if <i>uplo</i> = 'U': upper triangular part; the strictly lower triangular part of $\text{sub}(A)$ is not changed; if <i>uplo</i> = 'L': lower triangular part; the strictly upper triangular part of $\text{sub}(A)$ is not changed. Otherwise: all of the matrix $\text{sub}(A)$ is set.
<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).
<i>alpha</i>	(global). The constant to which the offdiagonal elements are to be set.
<i>beta</i>	(global). The constant to which the diagonal elements are to be set.

Output Parameters

<i>a</i>	(local). Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$ to be set. On exit, the leading m -by- n matrix $\text{sub}(A)$ is set as follows: if <i>uplo</i> = 'U', $A(ia+i-1, ja+j-1) = \alpha$, $1 \leq i \leq j-1$, $1 \leq j \leq n$, if <i>uplo</i> = 'L', $A(ia+i-1, ja+j-1) = \alpha$, $j+1 \leq i \leq m$, $1 \leq j \leq n$,
----------	--

otherwise, $A(ia+i-1, ja+j-1) = \alpha$, $1 \leq i \leq m$, $1 \leq j \leq n$, $ia+i \neq ja+j$, and, for all $uplo$, $A(ia+i-1, ja+i-1) = \beta$, $1 \leq i \leq \min(m, n)$.

ia, ja

(global)

The column and row indices in the distributed matrix *A* indicating the first row and column of the matrix sub(*A*), respectively.

desca

(global and local)

Array of size *dlen_*. The array descriptor for the distributed matrix *A*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lasmsub

Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.

Syntax

```
void pslasmsub (const float *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT
 *l, MKL_INT *k, const float *smlnum, float *buf, const MKL_INT *lwork );

void pdlasmsub (const double *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT
 *l, MKL_INT *k, const double *smlnum, double *buf, const MKL_INT *lwork );

void pclasmsub (const MKL_Complex8 *a , const MKL_INT *desca , const MKL_INT *i , const
MKL_INT *l , MKL_INT *k , const float *smlnum , MKL_Complex8 *buf , const MKL_INT
 *lwork );

void pzlasmsub (const MKL_Complex16 *a , const MKL_INT *desca , const MKL_INT *i ,
const MKL_INT *l , MKL_INT *k , const double *smlnum , MKL_Complex16 *buf , const
MKL_INT *lwork );
```

Include Files

- mkl_scalapack.h

Description

The p?lasmsub function looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero. This function performs a global maximum and must be called by all processes.

Input Parameters

a

(local)

Array of size *lld_a*LOCc(n_a)*.

On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.

desca

(global and local)

Array of size *dlen_*. The array descriptor for the distributed matrix *A*.

i

(global)

The global location of the bottom of the unreduced submatrix of *A*. Unchanged on exit.

<i>l</i>	(global) The global location of the top of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>smlnum</i>	(global) On entry, a "small number" for the given matrix. Unchanged on exit. The machine-dependent constants for the stopping criterion.
<i>lwork</i>	(local) This must be at least $2 * \text{ceil}(\text{ceil}((i-1)/mb_a) / \text{lcm}(nprow, npcol))$. Here <i>lcm</i> is least common multiple and <i>nprow</i> \times <i>npcol</i> is the logical grid size.

Output Parameters

<i>k</i>	(global) On exit, this yields the bottom portion of the unreduced submatrix. This will satisfy: $l \leq k \leq i-1$.
<i>buf</i>	(local). Array of size <i>lwork</i> .

Application Notes

This routine parallelizes the code from `?lahqr` that looks for a single small subdiagonal element.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lasrt

Sorts the numbers in an array and the corresponding vectors in increasing order.

Syntax

```
void pslasrt (const char* id, const MKL_INT* n, float* d, const float* q, const
MKL_INT* iq, const MKL_INT* jq, const MKL_INT* descq, float* work, const MKL_INT*
lwork, MKL_INT* iwork, const MKL_INT* liwork, MKL_INT* info);

void pdlasrt (const char* id, const MKL_INT* n, double* d, const double* q, const
MKL_INT* iq, const MKL_INT* jq, const MKL_INT* descq, double* work, const MKL_INT*
lwork, MKL_INT* iwork, const MKL_INT* liwork, MKL_INT* info);
```

Include Files

- `mkl_scalapack.h`

Description

`p?lasrt` sorts the numbers in *d* and the corresponding vectors in *q* in increasing order.

Input Parameters

<i>id</i>	(global) = 'I': sort <i>d</i> in increasing order;
-----------	---

	= 'D': sort d in decreasing order. (NOT IMPLEMENTED YET)
n	(global) The number of columns to be operated on i.e the number of columns of the distributed submatrix sub(Q). $n \geq 0$.
d	(global) Array, size (n)
q	(local) Pointer into the local memory to an array of size $lld_q * LOCC(jq+n-1)$. This array contains the local pieces of the distributed matrix sub(A) to be copied from.
iq	(global) The row index in the global array A indicating the first row of sub(Q).
jq	(global) The column index in the global array A indicating the first column of sub(Q).
$descq$	(global and local) Array of size $dlen_$. The array descriptor for the distributed matrix A .
$work$	(local) Array, size ($lwork$)
$lwork$	(local) The size of the array $work$. $lwork = \text{MAX}(n, NP * (NB + NQ))$, where $NP = \text{numroc}(n, NB, \text{MYROW}, \text{IAROW}, \text{NPROW})$, $NQ = \text{numroc}(n, NB, \text{MYCOL}, \text{DESCQ}(csrc_), \text{NPCOL})$. numroc is a ScaLAPACK tool function.
$iwork$	(local) Array, size ($liwork$)
$liwork$	(local) The size of the array $iwork$. $liwork = n + 2*NB + 2*NPCOL$

Output Parameters

d	On exit, the numbers in d are sorted in increasing order.
$info$	(global) = 0: successful exit

< 0: If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
void pslasq (MKL_INT *n , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , float *scale , float *sumsq );

void pdlasq (MKL_INT *n , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , double *scale , double *sumsq );

void pclasq (MKL_INT *n , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , float *scale , float *sumsq );

void pzlasq (MKL_INT *n , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , double *scale , double *sumsq );
```

Include Files

- mkl_scalapack.h

Description

The p?lassq function returns the values scl and ssq such that

$$scl^2 * ssq = x_1^2 + \dots + x_n^2 + scale^2 * sumsq,$$

where

$x_i = \text{sub}(X) = X(ix + (jx-1)*m_x + (i-1)*incx)$ for pslasq/pdlasq ,

$x_i = \text{sub}(X) = \text{abs}(X(ix + (jx-1)*m_x + (i-1)*incx))$ for pclasq/pzlasq.

For real functions pslasq/pdlasq the value of $sumsq$ is assumed to be non-negative and scl returns the value

$$scl = \max(scale, \text{abs}(x_i)).$$

For complex functions pclasq/pzlasq the value of $sumsq$ is assumed to be at least unity and the value of ssq will then satisfy

$$1.0 \leq ssq \leq sumsq + 2n$$

Value $scale$ is assumed to be non-negative and scl returns the value

$$scl = \max_i \left(scale, \text{abs}(\text{real}(x_i)), \text{abs}(\text{aimag}(x_i)) \right)$$

For all functions p?lassq values $scale$ and $sumsq$ must be supplied in $scale$ and $sumsq$ respectively, and $scale$ and $sumsq$ are overwritten by scl and ssq respectively.

All functions p?lassq make only one pass through the vector $\text{sub}(X)$.

Input Parameters

n (global)
The length of the distributed vector $\text{sub}(x)$.

<code>x</code>	<p>The array that stores the vector for which a scaled sum of squares is computed:</p> $x[ix + (jx-1)*m_x + i*incx], 0 \leq i < n.$
<code>ix</code>	<p>(global)</p> <p>The row index in the global matrix <i>X</i> indicating the first row of sub(<i>X</i>).</p>
<code>jx</code>	<p>(global)</p> <p>The column index in the global matrix <i>X</i> indicating the first column of sub(<i>X</i>).</p>
<code>descx</code>	<p>(global and local) array of size <code>dlen_</code>.</p> <p>The array descriptor for the distributed matrix <i>X</i>.</p>
<code>incx</code>	<p>(global)</p> <p>The global increment for the elements of <i>X</i>. Only two values of <i>incx</i> are supported in this version, namely 1 and <i>m_x</i>. The argument <i>incx</i> must not equal zero.</p>
<code>scale</code>	<p>(local).</p> <p>On entry, the value <i>scale</i> in the equation above.</p>
<code>sumsq</code>	<p>(local)</p> <p>On entry, the value <i>sumsq</i> in the equation above.</p>

Output Parameters

<code>scale</code>	<p>(local).</p> <p>On exit, <i>scale</i> is overwritten with <i>scl</i>, the scaling factor for the sum of squares.</p>
<code>sumsq</code>	<p>(local).</p> <p>On exit, <i>sumsq</i> is overwritten with the value <i>smsq</i>, the basic sum of squares from which <i>scl</i> has been factored out.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
void pslaswp (char *direc , char *rowcol , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *k1 , MKL_INT *k2 , MKL_INT *ipiv );

void pdlaswp (char *direc , char *rowcol , MKL_INT *n , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *k1 , MKL_INT *k2 , MKL_INT *ipiv );

void pclaswp (char *direc , char *rowcol , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *k1 , MKL_INT *k2 , MKL_INT *ipiv );

void pzlaswp (char *direc , char *rowcol , MKL_INT *n , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *k1 , MKL_INT *k2 , MKL_INT *ipiv );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?laswp` function performs a series of row or column interchanges on the distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$. One interchange is initiated for each of rows or columns $k1$ through $k2$ of $\text{sub}(A)$. This function assumes that the pivoting information has already been broadcast along the process row or column. Also note that this function will only work for $k1$ - $k2$ being in the same mb (or nb) block. If you want to pivot a full matrix, use `p?lapiv`.

Input Parameters

<code>direc</code>	(global) Specifies in which order the permutation is applied: = 'F' - forward, = 'B' - backward.
<code>rowcol</code>	(global) Specifies if the rows or columns are permuted: = 'R' - rows, = 'C' - columns.
<code>n</code>	(global) If <code>rowcol='R'</code> , the length of the rows of the distributed matrix $A(*, ja:ja+n-1)$ to be permuted; If <code>rowcol='C'</code> , the length of the columns of the distributed matrix $A(ia:ia+n-1, *)$ to be permuted;
<code>a</code>	(local) Pointer into the local memory to an array of size $lld_a * LOCc(n_a)$. On entry, this array contains the local pieces of the distributed matrix to which the row/columns interchanges will be applied.
<code>ia</code>	(global) The row index in the global matrix A indicating the first row of $\text{sub}(A)$.
<code>ja</code>	(global) The column index in the global matrix A indicating the first column of $\text{sub}(A)$.
<code>desca</code>	(global and local) array of size <code>dlen_</code> . The array descriptor for the distributed matrix A .
<code>k1</code>	(global) The first element of <code>ipiv</code> for which a row or column interchange will be done.
<code>k2</code>	(global) The last element of <code>ipiv</code> for which a row or column interchange will be done.

ipiv (local)
 Array of size $LOCr(m_a) + mb_a$ for row pivoting and $LOCr(n_a) + nb_a$ for column pivoting. This array is tied to the matrix *A*, *ipiv*[*k*]=1 implies rows (or columns) *k*+1 and *l* are to be interchanged, *k* = 0, 1, ..., size(*ipiv*) -1.

Output Parameters

A (local)
 On exit, the permuted distributed matrix.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?latra

Computes the trace of a general square distributed matrix.

Syntax

```
float pslatra (MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
double pdlatra (MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pclatra (MKL_Complex8 * , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pzlatra (MKL_Complex16 * , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
```

Include Files

- mkl_scalapack.h

Description

This function computes the trace of an *n*-by-*n* distributed matrix sub(*A*) denoting *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1). The result is left on every process of the grid.

Input Parameters

n (global)
 The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(*A*). *n* ≥ 0.

a (local).
 Pointer into the local memory to an array of size *lld_a* * *LOCc*(*ja*+*n*-1) containing the local pieces of the distributed matrix, the trace of which is to be computed.

ia, ja (global) The row and column indices respectively in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.

desca (global and local) array of size *dlen_*. The array descriptor for the distributed matrix *A*.

Output Parameters

val The value returned by the function.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

```
void pslatrd (char *uplo , MKL_INT *n , MKL_INT *nb , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *d , float *e , float *tau , float *w , MKL_INT *iw ,
MKL_INT *jw , MKL_INT *descw , float *work );

void pdlatrd (char *uplo , MKL_INT *n , MKL_INT *nb , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *d , double *e , double *tau , double *w , MKL_INT *iw ,
MKL_INT *jw , MKL_INT *descw , double *work );

void pclatrd (char *uplo , MKL_INT *n , MKL_INT *nb , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *d , float *e , MKL_Complex8 *tau , MKL_Complex8
*w , MKL_INT *iw , MKL_INT *jw , MKL_INT *descw , MKL_Complex8 *work );

void pzlatrd (char *uplo , MKL_INT *n , MKL_INT *nb , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *d , double *e , MKL_Complex16 *tau ,
MKL_Complex16 *w , MKL_INT *iw , MKL_INT *jw , MKL_INT *descw , MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description

The p?latrd function reduces nb rows and columns of a real symmetric or complex Hermitian matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ to symmetric/complex tridiagonal form by an orthogonal/unitary similarity transformation $Q' * \text{sub}(A) * Q$, and returns the matrices V and W , which are needed to apply the transformation to the unreduced part of $\text{sub}(A)$.

If $uplo = U$, p?latrd reduces the last nb rows and columns of a matrix, of which the upper triangle is supplied;

if $uplo = L$, p?latrd reduces the first nb rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary function called by [p?sytrd](#)/[p?hetrd](#).

Input Parameters

uplo (global)
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored:
= 'U': Upper triangular
= L: Lower triangular.

<i>n</i>	(global) The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$. $n \geq 0$.
<i>nb</i>	(global) The number of rows and columns to be reduced.
<i>a</i>	Pointer into the local memory to an array of size $ld_a * LOCC(j_a + n - 1)$. On entry, this array contains the local pieces of the symmetric/Hermitian distributed matrix $\text{sub}(A)$. If $uplo = U$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If $uplo = L$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.
<i>ia</i>	(global) The row index in the global matrix A indicating the first row of $\text{sub}(A)$.
<i>ja</i>	(global) The column index in the global matrix A indicating the first column of $\text{sub}(A)$.
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>iw</i>	(global) The row index in the global matrix W indicating the first row of $\text{sub}(W)$.
<i>jw</i>	(global) The column index in the global matrix W indicating the first column of $\text{sub}(W)$.
<i>descw</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix W .
<i>work</i>	(local) Workspace array of size nb_a .

Output Parameters

<i>a</i>	(local) On exit, if $uplo = 'U'$, the last nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of $\text{sub}(A)$; the elements above the diagonal with the array τ represent the orthogonal/unitary matrix Q as a product of elementary reflectors;
----------	---

if $uplo = 'L'$, the first nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of $sub(A)$; the elements below the diagonal with the array tau represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

 d

(local)

Array of size $LOCc(ja+n-1)$.

The diagonal elements of the tridiagonal matrix T : $d[i] = A(i+1, i+1)$, $i = 0, 1, \dots, LOCc(ja+n-1)-1$. d is tied to the distributed matrix A .

 e

(local)

Array of size $LOCc(ja+n-1)$ if $uplo = 'U'$, $LOCc(ja+n-2)$ otherwise.

The off-diagonal elements of the tridiagonal matrix T :

$$e[i] = A(i+1, i+2) \text{ if } uplo = 'U',$$

$$e[i] = A(i+2, i+1) \text{ if } uplo = 'L',$$

$$i = 0, 1, \dots, LOCc(ja+n-1)-1.$$

e is tied to the distributed matrix A .

 tau

(local)

Array of size $LOCc(ja+n-1)$. This array contains the scalar factors of the elementary reflectors. tau is tied to the distributed matrix A .

 w

(local)

Pointer into the local memory to an array of size $lld_w * nb_w$. This array contains the local pieces of the n -by- nb_w matrix w required to update the unreduced part of $sub(A)$.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n) * H(n-1) * \dots * H(n-nb+1)$$

Each $H(i)$ has the form

$$H(i) = I - tau * v * v',$$

where tau is a real/complex scalar, and v is a real/complex vector with $v(i:n) = 0$ and $v(i-1) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-1, ja+i)$, and tau in $tau[ja+i-2]$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - tau * v * v',$$

where tau is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and tau in $tau[ja+i-2]$.

The elements of the vectors v together form the n -by- nb matrix V which is needed, with W , to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank- $2k$ update of the form:

$$sub(A) := sub(A) - v w' - w v'.$$

The contents of a on exit are illustrated by the following examples with

$n = 5$ and $nb = 2$:

$$\begin{array}{l} \text{if } \text{uplo} = 'U': \\ \begin{bmatrix} a & a & a & v_4 & v_5 \\ & a & a & v_4 & v_5 \\ & & a & 1 & v_5 \\ & & & d & 1 \\ & & & & d \end{bmatrix} \end{array} \qquad \begin{array}{l} \text{if } \text{uplo} = 'L': \\ \begin{bmatrix} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_2 & a & a & \\ v_1 & v_2 & a & a & a \end{bmatrix} \end{array}$$

where d denotes a diagonal element of the reduced matrix, a denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining $H(i)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
void pslatrs (char *uplo , char *trans , char *diag , char *normin , MKL_INT *n , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , float *scale , float *cnorm , float *work );

void pdlatrs (char *uplo , char *trans , char *diag , char *normin , MKL_INT *n , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , double *scale , double *cnorm , double *work );

void pclatrs (char *uplo , char *trans , char *diag , char *normin , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *scale , float *cnorm , MKL_Complex8
*work );

void pzlatrs (char *uplo , char *trans , char *diag , char *normin , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *scale , double *cnorm ,
MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description

The `p?latrs` function solves a triangular system of equations $Ax = sb$, $A^T x = sb$ or $A^H x = sb$, where s is a scale factor set to prevent overflow. The description of the function will be extended in the future releases.

Input Parameters

<i>uplo</i>	<p>Specifies whether the matrix A is upper or lower triangular.</p> <p>= 'U': Upper triangular</p> <p>= 'L': Lower triangular</p>
<i>trans</i>	<p>Specifies the operation applied to Ax.</p> <p>= 'N': Solve $Ax = s*b$ (no transpose)</p> <p>= 'T': Solve $A^T x = s*b$ (transpose)</p> <p>= 'C': Solve $A^H x = s*b$ (conjugate transpose), where s - is a scale factor</p>
<i>diag</i>	<p>Specifies whether or not the matrix A is unit triangular.</p> <p>= 'N': Non-unit triangular</p> <p>= 'U': Unit triangular</p>
<i>normin</i>	<p>Specifies whether <i>cnorm</i> has been set or not.</p> <p>= 'Y': <i>cnorm</i> contains the column norms on entry;</p> <p>= 'N': <i>cnorm</i> is not set on entry. On exit, the norms will be computed and stored in <i>cnorm</i>.</p>
<i>n</i>	<p>The order of the matrix A. $n \geq 0$</p>
<i>a</i>	<p>Array of size $lda * n$. Contains the triangular matrix A.</p> <p>If <i>uplo</i> = 'U', the leading n-by-n upper triangular part of the array <i>a</i> contains the upper triangular matrix, and the strictly lower triangular part of <i>a</i> is not referenced.</p> <p>If <i>uplo</i> = 'L', the leading n-by-n lower triangular part of the array <i>a</i> contains the lower triangular matrix, and the strictly upper triangular part of <i>a</i> is not referenced.</p> <p>If <i>diag</i> = 'U', the diagonal elements of <i>a</i> are also not referenced and are assumed to be 1.</p>
<i>ia, ja</i>	<p>(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.</p>
<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix A.</p>
<i>x</i>	<p>Array of size n. On entry, the right hand side b of the triangular system.</p>
<i>ix</i>	<p>(global).The row index in the global matrix X indicating the first row of sub(x).</p>
<i>jx</i>	<p>(global)</p> <p>The column index in the global matrix X indicating the first column of sub(X).</p>

<i>descx</i>	(global and local) Array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>X</i> .
<i>cnorm</i>	Array of size <i>n</i> . If <i>normin</i> = 'Y', <i>cnorm</i> is an input argument and <i>cnorm</i> [<i>j</i>] contains the norm of the off-diagonal part of the (<i>j</i> +1)-th column of the matrix <i>A</i> , <i>j</i> =0, 1, ..., <i>n</i> -1. If <i>trans</i> = 'N', <i>cnorm</i> [<i>j</i>] must be greater than or equal to the infinity-norm, and if <i>trans</i> = 'T' or 'C', <i>cnorm</i> [<i>j</i>] must be greater than or equal to the 1-norm.
<i>work</i>	(local). Temporary workspace.

Output Parameters

<i>x</i>	On exit, <i>x</i> is overwritten by the solution vector <i>x</i> .
<i>scale</i>	Array of size <i>lda</i> * <i>n</i> . The scaling factor <i>s</i> for the triangular system as described above. If <i>scale</i> = 0, the matrix <i>A</i> is singular or badly scaled, and the vector <i>x</i> is an exact or approximate solution to $Ax = 0$.
<i>cnorm</i>	If <i>normin</i> = 'N', <i>cnorm</i> is an output argument and <i>cnorm</i> [<i>j</i>] returns the 1-norm of the off-diagonal part of the (<i>j</i> +1)-th column of <i>A</i> , <i>j</i> =0, 1, ..., <i>n</i> -1.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?latrz

Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.

Syntax

```
void pslatz (MKL_INT *m , MKL_INT *n , MKL_INT *l , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work );

void pdlatrz (MKL_INT *m , MKL_INT *n , MKL_INT *l , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work );

void pclatz (MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work );

void pzlatrz (MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work );
```

Include Files

- `mk1_scalapack.h`

Description

The `p?latrz` function reduces the *m*-by-*n* ($m \leq n$) real/complex upper trapezoidal matrix $\text{sub}(A) = [A(ia:ia+m-1, ja:ja+m-1)A(ia:ia+m-1, ja+n-l:ja+n-1)]$ to upper triangular form by means of orthogonal/unitary transformations.

The upper trapezoidal matrix $\text{sub}(A)$ is factored as

$\text{sub}(A) = \begin{pmatrix} R & 0 \end{pmatrix} * Z,$

where Z is an n -by- n orthogonal/unitary matrix and R is an m -by- m upper triangular matrix.

Input Parameters

m	(global) The number of rows in the distributed matrix $\text{sub}(A)$. $m \geq 0$.
n	(global) The number of columns in the distributed matrix $\text{sub}(A)$. $n \geq 0$.
l	(global) The number of columns of the distributed matrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. $l > 0$.
a	(local) Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$. On entry, the local pieces of the m -by- n distributed matrix $\text{sub}(A)$, which is to be factored.
ia	(global) The row index in the global matrix A indicating the first row of $\text{sub}(A)$.
ja	(global) The column index in the global matrix A indicating the first column of $\text{sub}(A)$.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
$work$	(local) Workspace array of size $lwork$. $lwork \geq nq0 + \max(1, mp0)$, where $iroff = \text{mod}(ia-1, mb_a),$ $icoff = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow),$ $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcot),$ $mp0 = \text{numroc}(m+iroff, mb_a, myrow, iarow, nprow),$ $nq0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npcot),$ numroc , indxg2p , and numroc are ScaLAPACK tool functions; $myrow$, $mycol$, $nprow$, and $npcot$ can be determined by calling the function <code>blacs_gridinfo</code> .

Output Parameters

<i>a</i>	On exit, the leading m -by- m upper triangular part of $\text{sub}(A)$ contains the upper triangular matrix R , and elements $n-l+1$ to n of the first m rows of $\text{sub}(A)$, with the array <i>tau</i> , represent the orthogonal/unitary matrix Z as a product of m elementary reflectors.
<i>tau</i>	(local) Array of size $\text{LOCr}(ja+m-1)$. This array contains the scalar factors of the elementary reflectors. <i>tau</i> is tied to the distributed matrix A .

Application Notes

The factorization is obtained by Householder's method. The k -th transformation matrix, $Z(k)$, which is used (or, in case of complex functions, whose conjugate transpose is used) to introduce zeros into the $(m - k + 1)$ -th row of $\text{sub}(A)$, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix},$$

where

$$T(k) = I - \tau \cdot u(k) \cdot u(k)', \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

τ is a scalar and $z(k)$ is an $(n-m)$ -element vector. τ and $z(k)$ are chosen to annihilate the elements of the k -th row of $\text{sub}(A)$. The scalar τ is returned in the k -th element of *tau*, indexed $k-1$, and the vector $u(k)$ in the k -th row of $\text{sub}(A)$, such that the elements of $z(k)$ are in $A(k, m+1), \dots, A(k, n)$. The elements of R are returned in the upper triangular part of $\text{sub}(A)$.

Z is given by

$$Z = Z(1)Z(2)\dots Z(m).$$

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lauu2

*Computes the product $U*U'$ or $L'*L$, where U and L are upper or lower triangular matrices (local unblocked algorithm).*

Syntax

```
void pslauu2 (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca );

void pdlauu2 (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca );

void pclauu2 (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca );
```

```
void pzlaau2 (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca );
```

Include Files

- mkl_scalapack.h

Description

The pzlaau2 function computes the product $U*U'$ or $L'*L$, where the triangular factor U or L is stored in the upper or lower triangular part of the distributed matrix

$\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

If $uplo = 'U'$ or $'u'$, then the upper triangle of the result is stored, overwriting the factor U in $\text{sub}(A)$.

If $uplo = 'L'$ or $'l'$, then the lower triangle of the result is stored, overwriting the factor L in $\text{sub}(A)$.

This is the unblocked form of the algorithm, calling [BLAS Level 2 Routines](#). No communication is performed by this function, the matrix to operate on should be strictly local to one process.

Input Parameters

$uplo$	(global) Specifies whether the triangular factor stored in the <i>matrix</i> $\text{sub}(A)$ is upper or lower triangular: = U: upper triangular = L: lower triangular.
n	(global) The number of rows and columns to be operated on, that is, the order of the triangular factor U or L . $n \geq 0$.
a	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$. On entry, the local pieces of the triangular factor U or L .
ia	(global) The row index in the global matrix A indicating the first row of $\text{sub}(A)$.
ja	(global) The column index in the global matrix A indicating the first column of $\text{sub}(A)$.
$desca$	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

a	(local) On exit, if $uplo = 'U'$, the upper triangle of the distributed matrix $\text{sub}(A)$ is overwritten with the upper triangle of the product $U*U'$; if $uplo = 'L'$, the lower triangle of $\text{sub}(A)$ is overwritten with the lower triangle of the product $L'*L$.
-----	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lauum

*Computes the product $U*U'$ or $L'*L$, where U and L are upper or lower triangular matrices.*

Syntax

```
void pslauum (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pdlauum (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pclauum (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pzlauum (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
```

Include Files

- mkl_scalapack.h

Description

The p?lauum function computes the product $U*U'$ or $L'*L$, where the triangular factor U or L is stored in the upper or lower triangular part of the matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

If $uplo = 'U'$ or $'u'$, then the upper triangle of the result is stored, overwriting the factor U in $\text{sub}(A)$. If $uplo = 'L'$ or $'l'$, then the lower triangle of the result is stored, overwriting the factor L in $\text{sub}(A)$.

This is the blocked form of the algorithm, calling Level 3 PBLAS.

Input Parameters

$uplo$	(global) Specifies whether the triangular factor stored in the matrix $\text{sub}(A)$ is upper or lower triangular: = 'U': upper triangular = 'L': lower triangular.
n	(global) The number of rows and columns to be operated on, that is, the order of the triangular factor U or L . $n \geq 0$.
a	(local) Pointer into the local memory to an array of size $ld_a * LOCc(ja+n-1)$. On entry, the local pieces of the triangular factor U or L .
ia	(global) The row index in the global matrix A indicating the first row of $\text{sub}(A)$.
ja	(global)

The column index in the global matrix A indicating the first column of $\text{sub}(A)$.

desca

(global and local) array of size *dlen_*. The array descriptor for the distributed matrix A .

Output Parameters

a

(local)

On exit, if *uplo* = 'U', the upper triangle of the distributed matrix $\text{sub}(A)$ is overwritten with the upper triangle of the product $U*U'$; if *uplo* = 'L', the lower triangle of $\text{sub}(A)$ is overwritten with the lower triangle of the product $L'*L$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lawil

Forms the Wilkinson transform.

Syntax

```
void pslawil (const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *m, const float *a,
const MKL_INT *desca, const float *h44, const float *h33, const float *h43h34, float
*v );
```

```
void pdlawil (const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *m, const double *a,
const MKL_INT *desca, const double *h44, const double *h33, const double *h43h34,
double *v );
```

```
void pclawil (const MKL_INT *ii , const MKL_INT *jj , const MKL_INT *m , const
MKL_Complex8 *a , const MKL_INT *desca , const MKL_Complex8 *h44 , const MKL_Complex8
*h33 , const MKL_Complex8 *h43h34 , MKL_Complex8 *v );
```

```
void pzlawil (const MKL_INT *ii , const MKL_INT *jj , const MKL_INT *m , const
MKL_Complex16 *a , const MKL_INT *desca , const MKL_Complex16 *h44 , const
MKL_Complex16 *h33 , const MKL_Complex16 *h43h34 , MKL_Complex16 *v );
```

Include Files

- mkl_scalapack.h

Description

The p?lawil function gets the transform given by *h44*, *h33*, and *h43h34* into *v* starting at row *m*.

Input Parameters

ii

(global)

Number of the process row which owns the matrix element $A(m+2, m+2)$.

jj

(global)

Number of the process column which owns the matrix element $A(m+2, m+2)$.

m

(global)

	On entry, the location from where the transform starts (row m). Unchanged on exit.
a	(local) Array of size $lld_a * LOCc(n_a)$. On entry, the Hessenberg matrix. Unchanged on exit.
$desca$	(global and local) Array of size $dlen_$. The array descriptor for the distributed matrix A . Unchanged on exit.
$h43h34$	(global) These three values are for the double shift QR iteration. Unchanged on exit.

Output Parameters

v	(global) Array of size 3 that contains the transform on output.
-----	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?org2l/p?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by p?geqlf (unblocked algorithm).

Syntax

```
void psorg2l (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorg2l (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

void pcung2l (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzung2l (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?org2l/p?ung2l function generates an m -by- n real/complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m :

$Q = H(k) * \dots * H(2) * H(1)$ as returned by p?geqlf.

Input Parameters

<i>m</i>	(global) The number of rows in the distributed submatrix <i>Q</i> . $m \geq 0$.
<i>n</i>	(global) The number of columns in the distributed submatrix <i>Q</i> . $m \geq n \geq 0$.
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix <i>Q</i> . $n \geq k \geq 0$.
<i>a</i>	Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$. On entry, the <i>j</i> -th column of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(j)$, $ja+n-k \leq j \leq ja+n-1$, as returned by p?geqlf in the <i>k</i> columns of its <i>distributed matrix</i> argument $A(ia:*, ja+n-k:ja+n-1)$.
<i>ia</i>	(global) The row index in the global matrix <i>A</i> indicating the first row of sub(<i>A</i>).
<i>ja</i>	(global) The column index in the global matrix <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) Array of size $LOCc(ja+n-1)$. <i>tau</i> [<i>j</i>] contains the scalar factor of the elementary reflector $H(j+1)$, $j = 0, 1, \dots, LOCc(ja+n-1)-1$, as returned by p?geqlf .
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq mpa0 + \max(1, nqa0)$, where $iroffa = \text{mod}(ia-1, mb_a),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow),$ $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcot),$ $mpa0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow),$ $nqa0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcot).$ indxg2p and numroc are ScaLAPACK tool functions; <i>myrow</i> , <i>mycol</i> , <i>nprow</i> , and <i>npcol</i> can be determined by calling the function <code>blacs_gridinfo</code> .

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<code>a</code>	On exit, this array contains the local pieces of the m -by- n distributed matrix Q .
<code>work</code>	On exit, <code>work[0]</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local). = 0: successful exit < 0: if the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then <code>info = - (i*100 + j)</code> , if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

[p?org2r/p?ung2r](#)

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by [p?geqrf](#) (unblocked algorithm).

Syntax

```
void psorg2r (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorg2r (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

void pcung2r (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzung2r (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The [p?org2r/p?ung2r](#) function generates an m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m :

$$Q = H(1)*H(2)*...*H(k)$$

as returned by [p?geqrf](#).

Input Parameters

<i>m</i>	(global) The number of rows in the distributed submatrix Q . $m \geq 0$.
<i>n</i>	(global) The number of columns in the distributed submatrix Q . $m \geq n \geq 0$.
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.
<i>a</i>	Pointer into the local memory to an array of size $ld_a * LOCc(ja+n-1)$ On entry, the j -th column of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqrf in the <i>k</i> columns of its <i>distributed matrix</i> argument $A(ia:*, ja:ja+k-1)$.
<i>ia</i>	(global) The row index in the global matrix A indicating the first row of sub(A).
<i>ja</i>	(global) The column index in the global matrix A indicating the first column of sub(A).
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>tau</i>	(local) Array of size $LOCc(ja+k-1)$. <i>tau[j]</i> contains the scalar factor of the elementary reflector $H(j+1)$, $j = 0, 1, \dots, LOCc(ja+k-1)-1$, as returned by p?geqrf . This array is tied to the distributed matrix A .
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq mpa0 + \max(1, nqa0)$, where $iroffa = \text{mod}(ia-1, mb_a), \quad icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow),$ $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcot),$ $mpa0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow),$ $nqa0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcot).$

`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	On exit, this array contains the local pieces of the m -by- n distributed matrix Q .
<code>work</code>	On exit, <code>work[0]</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local). = 0: successful exit < 0: if the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then <code>info = - (i*100 + j)</code> , if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orgl2/p?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf (unblocked algorithm).

Syntax

```
void psorgl2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
             *ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdorgl2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
             *ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcungl2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
             MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
             *lwork , MKL_INT *info );
void pzungl2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
             MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
             *lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?orgl2/p?ungl2` function generates a m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = H(k) * \dots * H(2) * H(1)$ (for real flavors),

$Q = (H(k))^H * \dots * (H(2))^H * (H(1))^H$ (for complex flavors) as returned by `p?gelqf`.

Input Parameters

<i>m</i>	(global) The number of rows in the distributed submatrix <i>Q</i> . $m \geq 0$.
<i>n</i>	(global) The number of columns in the distributed submatrix <i>Q</i> . $n \geq m \geq 0$.
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix <i>Q</i> . $m \geq k \geq 0$.
<i>a</i>	Pointer into the local memory to an array of size <i>lld_a</i> * <i>LOCc</i> (<i>ja</i> + <i>n</i> -1). On entry, the <i>i</i> -th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector <i>H</i> (<i>i</i>), $ia \leq i \leq ia+k-1$, as returned by <code>p?gelqf</code> in the <i>k</i> rows of its <i>distributed matrix</i> argument <i>A</i> (<i>ia</i> : <i>ia</i> + <i>k</i> -1, <i>ja</i> :*).
<i>ia</i>	(global) The row index in the global matrix <i>A</i> indicating the first row of sub(<i>A</i>).
<i>ja</i>	(global) The column index in the global matrix <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) Array of size <i>LOCr</i> (<i>ja</i> + <i>k</i> -1). <i>tau</i> [<i>j</i>] contains the scalar factor of the elementary reflectors <i>H</i> (<i>j</i> +1), $j = 0, 1, \dots, LOCr(ja+k-1)-1$, as returned by <code>p?gelqf</code> . This array is tied to the distributed matrix <i>A</i> .
<i>WORK</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq nqa0 + \max(1, mpa0)$, where $irowfa = \text{mod}(ia-1, mb_a),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow),$ $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcot),$ $mpa0 = \text{numroc}(m+irowfa, mb_a, myrow, iarow, nprow),$ $nqa0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcot).$

`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	On exit, this array contains the local pieces of the m -by- n distributed matrix Q .
<code>work</code>	On exit, <code>work[0]</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local) = 0: successful exit < 0: if the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then <code>info = - (i*100 + j)</code> , if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?org2/p?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by `p?gerqf` (unblocked algorithm).

Syntax

```
void psorg2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorg2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );

void pcungr2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzungr2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?org2/p?ungr2` function generates an m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$Q = H(1)*H(2)*...*H(k)$ (for real flavors);

$Q = (H(1))^H*(H(2))^H*...*(H(k))^H$ (for complex flavors) as returned by `p?gerqf`.

Input Parameters

<i>m</i>	(global) The number of rows in the distributed submatrix <i>Q</i> . $m \geq 0$.
<i>n</i>	(global) The number of columns in the distributed submatrix <i>Q</i> . $n \geq m \geq 0$.
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix <i>Q</i> . $m \geq k \geq 0$.
<i>a</i>	Pointer into the local memory to an array of size $lld_a * LOCr(ja+n-1)$. On entry, the <i>i</i> -th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia+m-k \leq i \leq ia+m-1$, as returned by <code>p?gerqf</code> in the <i>k</i> rows of its <i>distributed matrix</i> argument $A(ia+m-k:ia+m-1, ja:*)$.
<i>ia</i>	(global) The row index in the global matrix <i>A</i> indicating the first row of sub(<i>A</i>).
<i>ja</i>	(global) The column index in the global matrix <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) Array of size $LOCr(ja+m-1)$. <i>tau</i> [<i>j</i>] contains the scalar factor of the elementary reflectors $H(j+1)$, $j = 0, 1, \dots, LOCr(ja+m-1)-1$, as returned by <code>p?gerqf</code> . This array is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq nqa0 + \max(1, mpa0)$, where $irowfa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$, $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$, $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcot)$, $mpa0 = \text{numroc}(m+irowfa, mb_a, myrow, iarow, nprow)$, $nqa0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcot)$. <code>indxg2p</code> and <code>numroc</code> are ScaLAPACK tool functions; <i>myrow</i> , <i>mycol</i> , <i>nprow</i> , and <i>npcol</i> can be determined by calling the function <code>blacs_gridinfo</code> .

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a	On exit, this array contains the local pieces of the m -by- n distributed matrix Q .
$work$	On exit, $work[0]$ returns the minimal and optimal $lwork$.
$info$	(local) = 0: successful exit < 0: if the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then $info = -(i*100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orm2l/p?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf (unblocked algorithm).

Syntax

```
void psorm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );

void pcunm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- `mk1_scalapack.h`

Description

The `p?orm2l/p?unm2l` function overwrites the general real/complex m -by- n distributed matrix sub $(C)=C(ic:ic+m-1,jc:jc+n-1)$ with

$Q * \text{sub}(C)$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T * \text{sub}(C) / Q^H * \text{sub}(C)$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$\text{sub}(C) * Q$ if $side = 'R'$ and $trans = 'N'$, or

$\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ as returned by `p?geqlf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	(global) = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left, = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
<i>trans</i>	(global) = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	(global) The number of rows in the distributed matrix $\text{sub}(C)$. $m \geq 0$.
<i>n</i>	(global) The number of columns in the distributed matrix $\text{sub}(C)$. $n \geq 0$.
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q . If $side = 'L'$, $m \geq k \geq 0$; if $side = 'R'$, $n \geq k \geq 0$.
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOCc(ja+k-1)$. On entry, the j -th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqlf</code> in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. The argument $A(ia:*, ja:ja+k-1)$ is modified by the function but restored on exit. If $side = 'L'$, $lld_a \geq \max(1, LOCr(ia+m-1))$, if $side = 'R'$, $lld_a \geq \max(1, LOCr(ia+n-1))$.
<i>ia</i>	(global) The row index in the global matrix A indicating the first row of $\text{sub}(A)$.

<i>ja</i>	(global) The column index in the global matrix <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) Array of size $LOCc(ja+n-1)$. <i>tau</i> [<i>j</i>] contains the scalar factor of the elementary reflector $H(j+1)$, $j = 0, 1, \dots, LOCc(ja+n-1)-1$, as returned by p?geqlf . This array is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) Pointer into the local memory to an array of size $lld_c * LOCc(jc+n-1)$. On entry, the local pieces of the distributed matrix sub (<i>C</i>).
<i>ic</i>	(global) The row index in the global matrix <i>C</i> indicating the first row of sub(<i>C</i>).
<i>jc</i>	(global) The column index in the global matrix <i>C</i> indicating the first column of sub(<i>C</i>).
<i>descc</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) Workspace array of size <i>lwork</i> . On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least if <i>side</i> = 'L', $lwork \geq mpc0 + \max(1, nqc0)$, if <i>side</i> = 'R', $lwork \geq nqc0 + \max(\max(1, mpc0), \text{numroc}(\text{numroc}(n + icoffc, nb_a, 0, 0, npc0l), nb_a, 0, 0, lcmq)),$ where $lcmq = lcm / npc0l$, $lcm = iclm(nprow, npc0l)$, $iroffc = \text{mod}(ic-1, mb_c)$, $icoffc = \text{mod}(jc-1, nb_c)$, $icrow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow)$, $iccol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, npc0l)$, $Mqc0 = \text{numroc}(m + icoffc, nb_c, mycol, icrow, nprow)$, $Npc0 = \text{numroc}(n + iroffc, mb_c, myrow, iccol, npc0l)$,

`ilcm`, `indxg2p`, and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the function `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>c</code>	On exit, <code>c</code> is overwritten by $Q \cdot \text{sub}(C)$, or $Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q$, or $\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$
<code>work</code>	On exit, <code>work[0]</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local) = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> -1, had an illegal value, then <code>info = - (i*100 + j)</code> , if the <i>i</i> -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If `side = 'L'`, (`mb_a == mb_c` && `iroffa == iroffc` && `iarow == icrow`)

If `side = 'R'`, (`mb_a == nb_c` && `iroffa == iroffc`).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orm2r/p?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by p?geqrf (unblocked algorithm).

Syntax

```
void psorm2r (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );
```

```
void pdorm2r (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );
```



```

void pcunm2r (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunm2r (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );

```

Include Files

- mkl_scalapack.h

Description

The `p?orm2r/p?unm2r` function overwrites the general real/complex m -by- n distributed matrix sub(C)= $C(ic:ic+m-1, jc:jc+n-1)$ with

$Q \cdot \text{sub}(C)$ if `side` = 'L' and `trans` = 'N', or

$Q^T \cdot \text{sub}(C)$ / $Q^H \cdot \text{sub}(C)$ if `side` = 'L' and `trans` = 'T' (for real flavors) or `trans` = 'C' (for complex flavors), or

$\text{sub}(C) \cdot Q$ if `side` = 'R' and `trans` = 'N', or

$\text{sub}(C) \cdot Q^T$ / $\text{sub}(C) \cdot Q^H$ if `side` = 'R' and `trans` = 'T' (for real flavors) or `trans` = 'C' (for complex flavors).

where Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) \cdot \dots \cdot H(2) \cdot H(1)$ as returned by `p?geqrf`. Q is of order m if `side` = 'L' and of order n if `side` = 'R'.

Input Parameters

<code>side</code>	(global) = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left, = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
<code>trans</code>	(global) = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<code>m</code>	(global) The number of rows in the distributed matrix sub(C). $m \geq 0$.
<code>n</code>	(global) The number of columns in the distributed matrix sub(C). $n \geq 0$.
<code>k</code>	(global) The number of elementary reflectors whose product defines the matrix Q . If <code>side</code> = 'L', $m \geq k \geq 0$;

	if $side = 'R', n \geq k \geq 0$.
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_a * LOCc(ja+k-1)$.</p> <p>On entry, the j-th column of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqrf</code> in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. The argument $A(ia:*, ja:ja+k-1)$ is modified by the function but restored on exit.</p> <p>If $side = 'L', lld_a \geq \max(1, LOCr(ia+m-1))$, if $side = 'R', lld_a \geq \max(1, LOCr(ia+n-1))$.</p>
<i>ia</i>	<p>(global)</p> <p>The row index in the global matrix A indicating the first row of sub(A).</p>
<i>ja</i>	<p>(global)</p> <p>The column index in the global matrix A indicating the first column of sub(A).</p>
<i>desca</i>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .
<i>tau</i>	<p>(local)</p> <p>Array of size $LOCc(ja+k-1)$. $tau[j]$ contains the scalar factor of the elementary reflector $H(j+1)$, $j = 0, 1, \dots, LOCc(ja+k-1)-1$, as returned by <code>p?geqrf</code>. This array is tied to the distributed matrix A.</p>
<i>c</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_c * LOCc(jc+n-1)$.</p> <p>On entry, the local pieces of the distributed matrix sub (C).</p>
<i>ic</i>	<p>(global)</p> <p>The row index in the global matrix C indicating the first row of sub(C).</p>
<i>jc</i>	<p>(global)</p> <p>The column index in the global matrix C indicating the first column of sub(C).</p>
<i>descc</i>	<p>(global and local) array of size $dlen_$.</p> <p>The array descriptor for the distributed matrix C.</p>
<i>work</i>	<p>(local)</p> <p>Workspace array of size $lwork$.</p>
<i>lwork</i>	<p>(local or global)</p> <p>The size of the array <i>work</i>.</p> <p><i>lwork</i> is local input and must be at least</p> <p>if $side = 'L', lwork \geq mpc0 + \max(1, nqc0)$,</p>

```
if side = 'R', lwork ≥ nqc0 + max(max(1, mpc0), numroc(numroc(n
+icoffc, nb_a, 0, 0, npcol), nb_a, 0, 0, lcmq)),
```

where

```
lcmq = lcm/npcol,
lcm = iclm(nprow, npcol),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrc_c, npcol),
Mqc0 = numroc(m+icoffc, nb_c, mycol, icrow, nprow),
Npc0 = numroc(n+iroffc, mb_c, myrow, iccol, npcol),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; myrow, mycol,
nprow, and npcol can be determined by calling the function
blacs_gridinfo.
```

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by $Q \cdot \text{sub}(C)$, or $Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q$, or $\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$
<i>work</i>	On exit, <i>work</i> [0] returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> -1, had an illegal value, then <i>info</i> = - (<i>i</i> *100 + <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', (mb_a == mb_c) && (iroffa == iroffc) && (iarow == icrow).

If *side* = 'R', (mb_a == nb_c) && (iroffa == iroffc).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?orml2/p?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf (unblocked algorithm).

Syntax

```
void psorml2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorml2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );

void pcunml2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunml2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?orml2/p?unml2 function overwrites the general real/complex m -by- n distributed matrix sub $(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

$Q \cdot \text{sub}(C)$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$\text{sub}(C) \cdot Q$ if $side = 'R'$ and $trans = 'N'$, or

$\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ (for real flavors)

$Q = (H(k))^H * \dots * (H(2))^H * (H(1))^H$ (for complex flavors)

as returned by p?gelqf . Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

side (global)
 = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left,
 = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.

<i>trans</i>	<p>(global)</p> <p>= 'N': apply Q (no transpose)</p> <p>= 'T': apply Q^T (transpose, for real flavors)</p> <p>= 'C': apply Q^H (conjugate transpose, for complex flavors)</p>
<i>m</i>	<p>(global)</p> <p>The number of rows in the distributed matrix $\text{sub}(C)$. $m \geq 0$.</p>
<i>n</i>	<p>(global)</p> <p>The number of columns in the distributed matrix $\text{sub}(C)$. $n \geq 0$.</p>
<i>k</i>	<p>(global)</p> <p>The number of elementary reflectors whose product defines the matrix Q.</p> <p>If $side = 'L', m \geq k \geq 0$;</p> <p>if $side = 'R', n \geq k \geq 0$.</p>
<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size</p> <p>$lld_a * LOCc(ja+m-1)$ if $side='L'$,</p> <p>$lld_a * LOCc(ja+n-1)$ if $side='R'$,</p> <p>where $lld_a \geq \max(1, LOCr(ia+k-1))$.</p> <p>On entry, the i-th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. The argument $A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.</p>
<i>ia</i>	<p>(global)</p> <p>The row index in the global matrix A indicating the first row of $\text{sub}(A)$.</p>
<i>ja</i>	<p>(global)</p> <p>The column index in the global matrix A indicating the first column of $\text{sub}(A)$.</p>
<i>desca</i>	<p>(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A.</p>
<i>tau</i>	<p>(local)</p> <p>Array of size $LOCc(ia+k-1)$. $tau[i]$ contains the scalar factor of the elementary reflector $H(i+1)$, $i = 0, 1, \dots, LOCc(ja+k-1)-1$, as returned by p?gelqf. This array is tied to the distributed matrix A.</p>
<i>c</i>	<p>(local)</p> <p>Pointer into the local memory to an array of size $lld_c * LOCc(jc+n-1)$. On entry, the local pieces of the distributed matrix $\text{sub}(C)$.</p>
<i>ic</i>	<p>(global)</p> <p>The row index in the global matrix C indicating the first row of $\text{sub}(C)$.</p>

<i>jc</i>	(global) The column index in the global matrix <i>C</i> indicating the first column of <i>sub(C)</i> .
<i>descC</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least <pre> if side = 'L', lwork ≥ mqc0 + max(max(1, npc0), numroc(numroc(m +icoffc, mb_a, 0, 0, nprow), mb_a, 0, 0, lcmp)), if side = 'R', lwork ≥ npc0 + max(1, mqc0), where lcmp = lcm / nprow, lcm = iclm(nprow, npc0), iroffc = mod(ic-1, mb_c), icoffc = mod(jc-1, nb_c), icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow), iccol = indxg2p(jc, nb_c, mycol, csrc_c, npc0), Mpc0 = numroc(m+icoffc, mb_c, mycol, icrow, nprow), Nqc0 = numroc(n+iroffc, nb_c, myrow, iccol, npc0), ilcm, indxg2p and numroc are ScaLAPACK tool functions; myrow, mycol, nprow, and npc0 can be determined by calling the function blacs_gridinfo. </pre> If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla .

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by $Q \cdot \text{sub}(C)$, or $Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q$, or $\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$
<i>work</i>	On exit, <i>work</i> [0] returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> -1, had an illegal value, then <i>info</i> = - (<i>i</i> *100 + <i>j</i>),

if the i -th argument is a scalar and had an illegal value,
then $info = -i$.

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If $side = 'L'$, $(nb_a == mb_c \ \&\& \ icoffa == iroffc)$

If $side = 'R'$, $(nb_a == nb_c \ \&\& \ icoffa == icoffc \ \&\& \ iacol == iccol)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?ormr2/p?unmr2

Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by p?gerqf (unblocked algorithm).

Syntax

```
void psormr2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );

void pdormr2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );

void pcunmr2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunmr2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?ormr2/p?unmr2 function overwrites the general real/complex m -by- n distributed matrix sub $(C)=C(ic:ic+m-1, jc:jc+n-1)$ with

$Q \cdot \text{sub}(C)$ if $side = 'L'$ and $trans = 'N'$, or

$Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$\text{sub}(C) \cdot Q$ if $side = 'R'$ and $trans = 'N'$, or

$\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1)*H(2)*...*H(k)$ (for real flavors)

$Q = (H(1))^H*(H(2))^H*...*(H(k))^H$ (for complex flavors)

as returned by `p?gerqf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	(global) = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left, = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
<i>trans</i>	(global) = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	(global) The number of rows in the distributed matrix <code>sub(C)</code> . $m \geq 0$.
<i>n</i>	(global) The number of columns in the distributed matrix <code>sub(C)</code> . $n \geq 0$.
<i>k</i>	(global) The number of elementary reflectors whose product defines the matrix Q . If $side = 'L'$, $m \geq k \geq 0$; if $side = 'R'$, $n \geq k \geq 0$.
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja+m-1)$ if $side='L'$, $lld_a * LOCC(ja+n-1)$ if $side='R'$, where $lld_a \geq \max(1, LOCr(ia+k-1))$. On entry, the i -th row of the matrix stored in <i>a</i> must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by <code>p?gerqf</code> in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. The argument $A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.
<i>ia</i>	(global) The row index in the global matrix A indicating the first row of <code>sub(A)</code> .
<i>ja</i>	(global)

	The column index in the global matrix <i>A</i> indicating the first column of <i>sub(A)</i> .
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) Array of size $LOCc(ia+k-1)$. <i>tau</i> [<i>j</i>] contains the scalar factor of the elementary reflector $H(j+1)$, $j = 0, 1, \dots, LOCc(ja+k-1)-1$, as returned by p?gerqf . This array is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) Pointer into the local memory to an array of size $lld_c * LOCc(jc+n-1)$. On entry, the local pieces of the distributed matrix <i>sub(C)</i> .
<i>ic</i>	(global) The row index in the global matrix <i>C</i> indicating the first row of <i>sub(C)</i> .
<i>jc</i>	(global) The column index in the global matrix <i>C</i> indicating the first column of <i>sub(C)</i> .
<i>descc</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) Workspace array of size <i>lwork</i> .
<i>lwork</i>	(local or global) The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least <pre> if side = 'L', lwork ≥ mpc0 + max(max(1, nqc0), numroc(numroc(m +iroffc, mb_a, 0, 0, nprow), mb_a, 0, 0, lcmp)), if side = 'R', lwork ≥ nqc0 + max(1, mpc0), where lcmp = lcm/nprow, lcm = iclm(nprow, npcol), iroffc = mod(ic-1, mb_c), icoffc = mod(jc-1, nb_c), icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow), iccol = indxg2p(jc, nb_c, mycol, csrc_c, npcol), Mpc0 = numroc(m+iroffc, mb_c, myrow, icrow, nprow), Nqc0 = numroc(n+icoffc, nb_c, mycol, iccol, npcol), ilcm, indxg2p and numroc are ScaLAPACK tool functions; myrow, mycol, nprow, and npcol can be determined by calling the function blacs_gridinfo. </pre>

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

c	On exit, c is overwritten by $Q \cdot \text{sub}(C)$, or $Q^T \cdot \text{sub}(C) / Q^H \cdot \text{sub}(C)$, or $\text{sub}(C) \cdot Q$, or $\text{sub}(C) \cdot Q^T / \text{sub}(C) \cdot Q^H$
$work$	On exit, $work[0]$ returns the minimal and optimal $lwork$.
$info$	(local) = 0: successful exit < 0: if the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then $info = -(i \cdot 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If $side = 'L'$, $(nb_a == mb_c) \ \&\& \ (icoffa == iroffc)$.

If $side = 'R'$, $(nb_a == nb_c) \ \&\& \ (icoffa == icoffc) \ \&\& \ (iacol == iccol)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pbtrsv

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by [p?pbtrf](#).

Syntax

```
void pspbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
float *a , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb ,
float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pdpbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
double *a , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb ,
double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcpbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
```

```
void pzpbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The pzpbtrsv function solves a banded triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(jb:jb+n-1, 1:nrhs)$$

or

$$A(1:n, ja:ja+n-1)^T * X = B(jb:jb+n-1, 1:nrhs) \text{ for real flavors,}$$

$$A(1:n, ja:ja+n-1)^H * X = B(jb:jb+n-1, 1:nrhs) \text{ for complex flavors,}$$

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Cholesky factorization code pzpbtrf and is stored in $A(1:n, ja:ja+n-1)$ and af . The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to $uplo$.

The function pzpbtrf must be called first.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>uplo</i>	(global) Must be 'U' or 'L'. If <i>uplo</i> = 'U', upper triangle of $A(1:n, ja:ja+n-1)$ is stored; If <i>uplo</i> = 'L', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<i>trans</i>	(global) Must be 'N' or 'T' or 'C'. If <i>trans</i> = 'N', solve with $A(1:n, ja:ja+n-1)$; If <i>trans</i> = 'T' or 'C' for real flavors, solve with $A(1:n, ja:ja+n-1)^T$. If <i>trans</i> = 'C' for complex flavors, solve with conjugate transpose $(A(1:n, ja:ja+n-1))^H$.
<i>n</i>	(global) The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \geq 0$.
<i>bw</i>	(global) The number of subdiagonals in 'L' or 'U', $0 \leq bw \leq n-1$.
<i>nrhs</i>	(global) The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \geq 0$.

<i>a</i>	<p>(local)</p> <p>Pointer into the local memory to an array with the first size $lld_a \geq (bw + 1)$, stored in <i>desca</i>.</p> <p>On entry, this array contains the local pieces of the n-by-n symmetric banded distributed Cholesky factor L or $L^T * A(1:n, ja:ja+n-1)$.</p> <p>This local portion is stored in the packed banded format used in LAPACK. See the <i>Application Notes</i> below and the ScaLAPACK manual for more detail on the format of distributed matrices.</p>
<i>ja</i>	<p>(global) The index in the global in the global matrix A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).</p>
<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix A.</p> <p>If 1D type (<i>dtype_a</i> = 501), then $dlen \geq 7$;</p> <p>If 2D type (<i>dtype_a</i> = 1), then $dlen \geq 9$.</p> <p>Contains information on mapping of A to memory. (See ScaLAPACK manual for full description and options.)</p>
<i>b</i>	<p>(local)</p> <p>Pointer into the local memory to an array of local lead size $lld_b \geq nb$.</p> <p>On entry, this array contains the local pieces of the right hand sides $B(jb:jb+n-1, 1:nrhs)$.</p>
<i>ib</i>	<p>(global) The row index in the global matrix B that points to the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).</p>
<i>descb</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix B.</p> <p>If 1D type (<i>dtype_b</i> = 502), then $dlen \geq 7$;</p> <p>If 2D type (<i>dtype_b</i> = 1), then $dlen \geq 9$.</p> <p>Contains information on mapping of B to memory. Please, see ScaLAPACK manual for full description and options.</p>
<i>laf</i>	<p>(local)</p> <p>The size of user-input auxiliary fill-in space <i>af</i>. Must be $laf \geq (nb + 2*bw) * bw$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i>[0].</p>
<i>work</i>	<p>(local)</p> <p>The array <i>work</i> is a temporary workspace array of size <i>lwork</i>. This space may be overwritten in between function calls.</p>
<i>lwork</i>	<p>(local or global) The size of the user-input workspace <i>work</i>, must be at least $lwork \geq bw * nrhs$. If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i>[0] and an error code is returned.</p>

Output Parameters

<i>af</i>	(local) The array <i>af</i> is of size <i>laf</i> . It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function p?pbtrf and is stored in <i>af</i> . If a linear system is to be solved using p?pbtrs after the factorization function, <i>af</i> must not be altered after the factorization.
<i>b</i>	On exit, this array contains the local piece of the solutions distributed matrix <i>X</i> .
<i>work</i> [0]	On exit, <i>work</i> [0] contains the minimum value of <i>lwork</i> .
<i>info</i>	(local) = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry, indexed <i>j</i> -1, had an illegal value, then <i>info</i> = - (<i>i</i> *100 + <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Application Notes

If the factorization function and the solve function are to be called separately to solve various sets of right-hand sides using the same coefficient matrix, the auxiliary space *af* must not be altered between calls to the factorization function and the solve function.

The best algorithm for solving banded and tridiagonal linear systems depends on a variety of parameters, especially the bandwidth. Currently, only algorithms designed for the case $N/P \gg bw$ are implemented. These algorithms go by many names, including Divide and Conquer, Partitioning, domain decomposition-type, etc.

The Divide and Conquer algorithm assumes the matrix is narrowly banded compared with the number of equations. In this situation, it is best to distribute the input matrix *A* one-dimensionally, with columns atomic and rows divided amongst the processes. The basic algorithm divides the banded matrix up into *P* pieces with one stored on each processor, and then proceeds in 2 phases for the factorization or 3 for the solution of a linear system.

- 1. Local Phase:** The individual pieces are factored independently and in parallel. These factors are applied to the matrix creating fill-in, which is stored in a non-inspectable way in auxiliary space *af*. Mathematically, this is equivalent to reordering the matrix *A* as PAP^T and then factoring the principal leading submatrix of size equal to the sum of the sizes of the matrices factored on each processor. The factors of these submatrices overwrite the corresponding parts of *A* in memory.
- 2. Reduced System Phase:** A small ($bw*(P-1)$) system is formed representing interaction of the larger blocks and is stored (as are its factors) in the space *af*. A parallel Block Cyclic Reduction algorithm is used. For a linear system, a parallel front solve followed by an analogous backsolve, both using the structure of the factored matrix, are performed.
- 3. Back Substitution Phase:** For a linear system, a local backsubstitution is performed on each processor in parallel.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?pttrsv

Solves a single triangular linear system via *frontsolve* or *backsolve* where the triangular matrix is a factor of a tridiagonal matrix computed by `p?pttrf`.

Syntax

```
void pspttrsv (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *d , float *e , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *af , MKL_INT
*laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pdpttrsv (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *d , double *e , MKL_INT
*ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *af , MKL_INT
*laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcpttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , float *d ,
MKL_Complex8 *e , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzpttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , double *d ,
MKL_Complex16 *e , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?pttrsv` function solves a tridiagonal triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(jb:jb+n-1, 1:nrhs)$$

or

$$A(1:n, ja:ja+n-1)^T * X = B(jb:jb+n-1, 1:nrhs) \text{ for real flavors,}$$

$$A(1:n, ja:ja+n-1)^H * X = B(jb:jb+n-1, 1:nrhs) \text{ for complex flavors,}$$

where $A(1:n, ja:ja+n-1)$ is a tridiagonal triangular matrix factor produced by the Cholesky factorization code `p?pttrf` and is stored in $A(1:n, ja:ja+n-1)$ and `af`. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to `uplo`.

The function `p?pttrf` must be called first.

Input Parameters

<code>uplo</code>	(global) Must be 'U' or 'L'. If <code>uplo</code> = 'U', upper triangle of $A(1:n, ja:ja+n-1)$ is stored; If <code>uplo</code> = 'L', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<code>trans</code>	(global) Must be 'N' or 'C'. If <code>trans</code> = 'N', solve with $A(1:n, ja:ja+n-1)$; If <code>trans</code> = 'C' (for complex flavors), solve with conjugate transpose $(A(1:n, ja:ja+n-1))^H$.

<i>n</i>	(global) The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \geq 0$.
<i>nrhs</i>	(global) The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \geq 0$.
<i>d</i>	(local) Pointer to the local part of the global vector storing the main diagonal of the matrix; must be of size $\geq nb_a$.
<i>e</i>	(local) Pointer to the local part of the global vector <i>du</i> storing the upper diagonal of the matrix; must be of size $\geq nb_a$. Globally, $du(n)$ is not referenced, and <i>du</i> must be aligned with <i>d</i> .
<i>ja</i>	(global) The index in the global matrix <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . If 1D type (<i>dtype_a</i> = 501 or 502), then $dlen \geq 7$; If 2D type (<i>dtype_a</i> = 1), then $dlen \geq 9$. Contains information on mapping of <i>A</i> to memory. See ScaLAPACK manual for full description and options.
<i>b</i>	(local) Pointer into the local memory to an array of local lead size $lld_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(jb:jb+n-1, 1:nrhs)$.
<i>ib</i>	(global) The row index in the global matrix <i>B</i> that points to the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> . If 1D type (<i>dtype_b</i> = 502), then $dlen \geq 7$; If 2D type (<i>dtype_b</i> = 1), then $dlen \geq 9$. Contains information on mapping of <i>B</i> to memory. See ScaLAPACK manual for full description and options.
<i>laf</i>	(local) The size of user-input auxiliary fill-in space <i>af</i> . Must be $laf \geq (nb + 2 * bw) * bw$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> [0].
<i>work</i>	(local)

The array *work* is a temporary workspace array of size *lwork*. This space may be overwritten in between function calls.

lwork

(local or global) The size of the user-input workspace *work*, must be at least $lwork \geq (10 + 2 * \min(100, nrhs)) * npcol + 4 * nrhs$. If *lwork* is too small, the minimal acceptable size will be returned in *work*[0] and an error code is returned.

Output Parameters

d, e

(local).

On exit, these arrays contain information on the factors of the matrix.

af

(local)

The array *af* is of size *laf*. It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function [p?pbtrf](#) and is stored in *af*. If a linear system is to be solved using [p?pttrs](#) after the factorization function, *af* must not be altered after the factorization.

b

On exit, this array contains the local piece of the solutions distributed matrix *X*.

work[0]

On exit, *work*[0] contains the minimum value of *lwork*.

info

(local)

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry, indexed *j*-1, had an illegal value,

then *info* = - (*i**100 + *j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?potf2

Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).

Syntax

```
void pspotf2 (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *info );

void pdpotf2 (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *info );

void pcspotf2 (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );

void pzpotf2 (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
```


Include Files

- `mk1_scalapack.h`

Description

The `p?potf2` function computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

The factorization has the form

$\text{sub}(A) = U^*U$, if $uplo = 'U'$, or $\text{sub}(A) = L^*L'$, if $uplo = 'L'$,

where U is an upper triangular matrix, L is lower triangular. X' denotes transpose (conjugate transpose) of X .

Input Parameters

<code>uplo</code>	(global) Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix A is stored. = <code>'U'</code> : upper triangle of $\text{sub}(A)$ is stored; = <code>'L'</code> : lower triangle of $\text{sub}(A)$ is stored.
<code>n</code>	(global) The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$. $n \geq 0$.
<code>a</code>	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$ containing the local pieces of the n -by- n symmetric distributed matrix $\text{sub}(A)$ to be factored. If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular matrix and the strictly lower triangular part of this matrix is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<code>ia, ja</code>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the $\text{sub}(A)$, respectively.
<code>desca</code>	(global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

<code>a</code>	(local) On exit, if $uplo = 'U'$, the upper triangular part of the distributed matrix contains the Cholesky factor U ; if $uplo = 'L'$, the lower triangular part of the distributed matrix contains the Cholesky factor L .
----------------	---

info (local)

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-th entry, indexed *j*-1, had an illegal value,

then *info* = - (*i**100 + *j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

> 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?rot

Applies a planar rotation to two distributed vectors.

Syntax

```
void psrot(MKL_INT* n, float* x, MKL_INT* ix, MKL_INT* jx, MKL_INT* descx, MKL_INT* incx, float* y, MKL_INT* iy, MKL_INT* jy, MKL_INT* descy, MKL_INT* incy, float* cs, float* sn, float* work, MKL_INT* lwork, MKL_INT* info);
```

```
void pdrot(MKL_INT* n, double* x, MKL_INT* ix, MKL_INT* jx, MKL_INT* descx, MKL_INT* incx, double* y, MKL_INT* iy, MKL_INT* jy, MKL_INT* descy, MKL_INT* incy, double* cs, double* sn, double* work, MKL_INT* lwork, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

p?rot applies a planar rotation defined by *cs* and *sn* to the two distributed vectors sub(*x*) and sub(*y*).

Input Parameters

<i>n</i>	(global) The number of elements to operate on when applying the planar rotation to <i>x</i> and <i>y</i> (<i>n</i> ≥0).
<i>x</i>	(local) array of size ((<i>jx</i> -1)* <i>m_x</i> + <i>ix</i> + (<i>n</i> - 1)*abs(<i>incx</i>)) This array contains the entries of the distributed vector sub(<i>x</i>).
<i>ix</i>	(global) The global row index of the submatrix of the distributed matrix <i>x</i> to operate on. If <i>incx</i> = 1, then it is required that <i>ix</i> = <i>iy</i> . 1 ≤ <i>ix</i> ≤ <i>m_x</i> .
<i>jx</i>	(global) The global column index of the submatrix of the distributed matrix <i>x</i> to operate on. If <i>incx</i> = <i>m_x</i> , then it is required that <i>jx</i> = <i>jy</i> . 1 ≤ <i>ix</i> ≤ <i>n_x</i> .
<i>descx</i>	(global and local) array of size 9

	The array descriptor of the distributed matrix x .
<i>incx</i>	(global) The global increment for the elements of x . Only two values of <i>incx</i> are supported in this version, namely 1 and m_x . Moreover, it must hold that $incx = m_x$ if $incy = m_y$ and that $incx = 1$ if $incy = 1$.
<i>y</i>	(local) array of size $((jy-1)*m_y + iy + (n-1)*abs(incy))$ This array contains the entries of the distributed vector sub(y).
<i>iy</i>	(global) The global row index of the submatrix of the distributed matrix y to operate on. If $incy = 1$, then it is required that $iy = ix$. $1 \leq iy \leq m_y$.
<i>jy</i>	(global) The global column index of the submatrix of the distributed matrix y to operate on. If $incy = m_x$, then it is required that $jy = jx$. $1 \leq jy \leq m_y$.
<i>descy</i>	(global and local) array of size 9 The array descriptor of the distributed matrix y .
<i>incy</i>	(global) The global increment for the elements of y . Only two values of <i>incy</i> are supported in this version, namely 1 and m_y . Moreover, it must hold that $incy = m_y$ if $incx = m_x$ and that $incy = 1$ if $incx = 1$.
<i>cs, sn</i>	(global) The parameters defining the properties of the planar rotation. It must hold that $0 \leq cs, sn \leq 1$ and that $sn^2 + cs^2 = 1$. The latter is hardly checked in finite precision arithmetics.
<i>work</i>	(local workspace) array of size <i>lwork</i>
<i>lwork</i>	(local) The length of the workspace array <i>work</i> . If $incx = 1$ and $incy = 1$, then $lwork = 2*m_x$ If $lwork = -1$, then a workspace query is assumed; the function only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by pxerbla .

OUTPUT Parameters

x	
y	
<i>work</i> [0]	On exit, if <i>info</i> = 0, <i>work</i> [0] returns the optimal <i>lwork</i>
<i>info</i>	(global) = 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value.

If the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?rscl

Multiplies a vector by the reciprocal of a real scalar.

Syntax

```
void psrscl (MKL_INT *n , float *sa , float *sx , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );

void pdrsc1 (MKL_INT *n , double *sa , double *sx , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );

void pcsrscl (MKL_INT *n , float *sa , MKL_Complex8 *sx , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );

void pzdrsc1 (MKL_INT *n , double *sa , MKL_Complex16 *sx , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
```

Include Files

- mkl_scalapack.h

Description

The `p?rscl` function multiplies an n -element real/complex vector $\text{sub}(X)$ by the real scalar $1/a$. This is done without overflow or underflow as long as the final result $\text{sub}(X)/a$ does not overflow or underflow.

$\text{sub}(X)$ denotes $X(ix:ix+n-1, jx:jx)$, if $incx = 1$,

and $X(ix:ix, jx:jx+n-1)$, if $incx = m_x$.

Input Parameters

n	(global) The number of components of the distributed vector $\text{sub}(X)$. $n \geq 0$.
sa	The scalar a that is used to divide each component of the vector $\text{sub}(X)$. This parameter must be ≥ 0 .
sx	Array containing the local pieces of a distributed matrix of size of at least $((jx-1)*m_x + ix + (n-1)*abs(incx))$. This array contains the entries of the distributed vector $\text{sub}(X)$.
ix	(global) The row index of the submatrix of the distributed matrix X to operate on.
jx	(global) The column index of the submatrix of the distributed matrix X to operate on.
$descx$	(global and local) Array of size 9. The array descriptor for the distributed matrix X .

incx (global)
The increment for the elements of *X*. This version supports only two values of *incx*, namely 1 and *m_x*.

Output Parameters

sx On exit, the result x/a .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?sygs2/p?hegs2

Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).

Syntax

```
void pssygs2 (MKL_INT *ibtype , char *uplo , MKL_INT *n , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_INT *info );

void pdsygs2 (MKL_INT *ibtype , char *uplo , MKL_INT *n , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_INT *info );

void pchehs2 (MKL_INT *ibtype , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );

void pzhehs2 (MKL_INT *ibtype , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The p?sygs2/p?hegs2 function reduces a real symmetric-definite or a complex Hermitian positive-definite generalized eigenproblem to standard form.

Here $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$, and $\text{sub}(B)$ denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If *ibtype* = 1, the problem is

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x$$

and $\text{sub}(A)$ is overwritten by

$\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$ or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$ - for real flavors, and

$\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$ or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$ - for complex flavors.

If *ibtype* = 2 or 3, the problem is

$$\text{sub}(A) * \text{sub}(B) x = \lambda * x \text{ or } \text{sub}(B) * \text{sub}(A) x = \lambda * x$$

and $\text{sub}(A)$ is overwritten by

$U * \text{sub}(A) * U^T$ or $L * * T * \text{sub}(A) * L$ for real flavors and

$U * \text{sub}(A) * U^H$ or $L * * H * \text{sub}(A) * L$ for complex flavors.

The matrix $\text{sub}(B)$ must have been previously factorized as $U^T * U$ or $L * L^T$ (for real flavors), or as $U^H * U$ or $L * L^H$ (for complex flavors) by `p?potrf`.

Input Parameters

<i>ibtype</i>	(global) = 1: compute $\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$ for real functions, and $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$ for complex functions; = 2 or 3: compute $U * \text{sub}(A) * U^T$, or $L^T * \text{sub}(A) * L$ for real functions, and $U * \text{sub}(A) * U^H$ or $L^H * \text{sub}(A) * L$ for complex functions.
<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored, and how $\text{sub}(B)$ is factorized. = 'U': Upper triangular of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factorized as $U^T * U$ (for real functions) or as $U^H * U$ (for complex functions). = 'L': Lower triangular of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factorized as $L * L^T$ (for real functions) or as $L * L^H$ (for complex functions)
<i>n</i>	(global) The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$. $n \geq 0$.
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOCC(ja+n-1)$. On entry, this array contains the local pieces of the n -by- n symmetric/Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>B</i>	(local) Pointer into the local memory to an array of size $lld_b * LOCC(jb+n-1)$.

On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of $\text{sub}(B)$ as returned by `p?potrf`.

ib, jb

(global)

The row and column indices in the global matrix B indicating the first row and the first column of the $\text{sub}(B)$, respectively.

descb

(global and local) array of size *dlen_*. The array descriptor for the distributed matrix B .

Output Parameters

a

(local)

On exit, if *info* = 0, the transformed matrix is stored in the same format as $\text{sub}(A)$.

info

= 0: successful exit.

< 0: if the *i*-th argument is an array and the *j*-th entry, indexed *j*-1, had an illegal value,

then *info* = - (*i**100+ *j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?sytd2/p?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).

Syntax

```
void pssytd2 (char *uplo, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT
*desca, float *d, float *e, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);

void pdsytd2 (char *uplo, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT
*desca, double *d, double *e, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);

void pchetd2 (char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT
*desca, float *d, float *e, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork,
MKL_INT *info);

void pzhetd2 (char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja,
MKL_INT *desca, double *d, double *e, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT
*lwork, MKL_INT *info);
```

Include Files

- `mk1_scalapack.h`

Description

The `p?sytd2/p?hetd2` function reduces a real symmetric/complex Hermitian matrix $\text{sub}(A)$ to symmetric/Hermitian tridiagonal form T by an orthogonal/unitary similarity transformation:

$Q^* \text{sub}(A) * Q = T$, where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored: = 'U': upper triangular = 'L': lower triangular
<i>n</i>	(global) The number of rows and columns to be operated on, that is, the order of the distributed matrix $\text{sub}(A)$. $n \geq 0$.
<i>a</i>	(local) Pointer into the local memory to an array of size $lld_a * LOC_c(ja+n-1)$. On entry, this array contains the local pieces of the n -by- n symmetric/Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<i>ia, ja</i>	(global) The row and column indices in the global matrix A indicating the first row and the first column of the $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix A .
<i>work</i>	(local) The array <i>work</i> is a temporary workspace array of size <i>lwork</i> .

Output Parameters

<i>a</i>	On exit, if <i>uplo</i> = 'U', the diagonal and first superdiagonal of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements above the first superdiagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors; if <i>uplo</i> = 'L', the diagonal and first subdiagonal of A are overwritten by the corresponding elements of the tridiagonal matrix T , and the elements below the first subdiagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors. See the <i>Application Notes</i> below.
<i>d</i>	(local) Array of size $LOC_c(ja+n-1)$. The diagonal elements of the tridiagonal matrix T :

	$d[i] = A(i+1, i+1)$, where $i=0, 1, \dots, LOCC(ja+n-1) - 1$; d is tied to the distributed matrix A .
<i>e</i>	<p>(local)</p> <p>Array of size $LOCC(ja+n-1)$,</p> <p>if $uplo = 'U'$, $LOCC(ja+n-2)$ otherwise.</p> <p>The off-diagonal elements of the tridiagonal matrix T:</p> <p>$e[i] = A(i+1, i+2)$ if $uplo = 'U'$,</p> <p>$e[i] = A(i+2, i+1)$ if $uplo = 'L'$,</p> <p>where $i=0, 1, \dots, LOCC(ja+n-1) - 1$.</p> <p>$e$ is tied to the distributed matrix A.</p>
<i>tau</i>	<p>(local)</p> <p>Array of size $LOCC(ja+n-1)$.</p> <p>The scalar factors of the elementary reflectors. tau is tied to the distributed matrix A.</p>
<i>work</i> [0]	On exit, <i>work</i> [0] returns the minimal and optimal value of <i>lwork</i> .
<i>lwork</i>	<p>(local or global)</p> <p>The size of the workspace array <i>work</i>.</p> <p><i>lwork</i> is local input and must be at least $lwork \geq 3n$.</p> <p>If $lwork = -1$, then <i>lwork</i> is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.</p>
<i>info</i>	<p>(local)</p> <p>= 0: successful exit</p> <p>< 0: if the i-th argument, indexed $i-1$, is an array and the j-th entry had an illegal value,</p> <p>then $info = -(i*100+j)$,</p> <p>if the i-th argument is a scalar and had an illegal value,</p> <p>then $info = -i$.</p>

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) * \dots * H(2) * H(1)$$

Each $H(i)$ has the form

$$H(i) = I - tau * v * v',$$

where tau is a real/complex scalar, and v is a real/complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and tau in $tau[ja+i-2]$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$Q = H(1) * H(2) * \dots * H(n-1).$

Each $H(i)$ has the form

$H(i) = I - \tau v v^T,$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and τ in $\tau[ja+i-2]$.

The contents of sub (A) on exit are illustrated by the following examples with $n = 5$:

$$\begin{array}{cc} \text{if uplo='U':} & \text{if uplo='L':} \\ \begin{bmatrix} d & e & v_2 & v_3 & v_4 \\ & d & e & v_3 & v_4 \\ & & d & e & v_4 \\ & & & d & e \\ & & & & d \end{bmatrix} & \begin{bmatrix} d & & & & \\ e & d & & & \\ v_1 & e & d & & \\ v_1 & v_2 & e & d & \\ v_1 & v_2 & v_3 & e & d \end{bmatrix} \end{array}$$

where d and e denotes diagonal and off-diagonal elements of T , and v_i denotes an element of the vector defining $H(i)$.

NOTE

The distributed matrix sub(A) must verify some alignment properties, namely the following expression should be true:

$(mb_a == nb_a \ \&\& \ iroffa == icoffa)$ where $iroffa = \text{mod}(ia - 1, mb_a)$ and $icoffa = \text{mod}(ja - 1, nb_a)$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trord

Reorders the Schur factorization of a general matrix.

Syntax

```
void pstrord( char* compq, MKL_INT* select, MKL_INT* para, MKL_INT* n, float* t,
MKL_INT* it, MKL_INT* jt, MKL_INT* desct, float* q, MKL_INT* iq, MKL_INT* jq, MKL_INT*
descq, float* wr, float* wi, MKL_INT* m, float* work, MKL_INT* lwork, MKL_INT* iwork,
MKL_INT* liwork, MKL_INT* info);

void pdtrord(char* compq, MKL_INT* select, MKL_INT* para, MKL_INT* n, double* t,
MKL_INT* it, MKL_INT* jt, MKL_INT* desct, double* q, MKL_INT* iq, MKL_INT* jq, MKL_INT*
descq, double* wr, double* wi, MKL_INT* m, double* work, MKL_INT* lwork, MKL_INT* iwork,
MKL_INT* liwork, MKL_INT* info);
```

Include Files

- `mkl_scalapack.h`

Description

`p?trord` reorders the real Schur factorization of a real matrix $A = Q * T * Q^T$, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T , and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace.

T must be in Schur form (as returned by [p?lahqr](#)), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.

This function uses a delay and accumulate procedure for performing the off-diagonal updates.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>compq</i>	(global) = 'V': update the matrix q of Schur vectors; = 'N': do not update q .												
<i>select</i>	(global) array of size n <i>select</i> specifies the eigenvalues in the selected cluster. To select a real eigenvalue $w(j)$, <i>select</i> [$j-1$] must be set to 1. To select a complex conjugate pair of eigenvalues $w(j)$ and $w(j+1)$, corresponding to a 2-by-2 diagonal block, either <i>select</i> [$j-1$] or <i>select</i> [j] or both must be set to 1; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.												
<i>para</i>	(global) Block parameters: <table> <tr> <td><i>para</i>[0]</td><td>maximum number of concurrent computational windows allowed in the algorithm; $0 < para[0] \leq \min(nprow, npcot)$ must hold;</td></tr> <tr> <td><i>para</i>[1]</td><td>number of eigenvalues in each window; $0 < para[1] < para[2]$ must hold;</td></tr> <tr> <td><i>para</i>[2]</td><td>window size; $para[1] < para[2] < mb_t$ must hold;</td></tr> <tr> <td><i>para</i>[3]</td><td>minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; $0 \leq para[3] \leq 100$ must hold;</td></tr> <tr> <td><i>para</i>[4]</td><td>width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; $0 < para[4] \leq mb_t$ must hold.</td></tr> <tr> <td><i>para</i>[5]</td><td>the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size; $0 < para[5] \leq para[1]$ must hold.</td></tr> </table>	<i>para</i> [0]	maximum number of concurrent computational windows allowed in the algorithm; $0 < para[0] \leq \min(nprow, npcot)$ must hold;	<i>para</i> [1]	number of eigenvalues in each window; $0 < para[1] < para[2]$ must hold;	<i>para</i> [2]	window size; $para[1] < para[2] < mb_t$ must hold;	<i>para</i> [3]	minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; $0 \leq para[3] \leq 100$ must hold;	<i>para</i> [4]	width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; $0 < para[4] \leq mb_t$ must hold.	<i>para</i> [5]	the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size; $0 < para[5] \leq para[1]$ must hold.
<i>para</i> [0]	maximum number of concurrent computational windows allowed in the algorithm; $0 < para[0] \leq \min(nprow, npcot)$ must hold;												
<i>para</i> [1]	number of eigenvalues in each window; $0 < para[1] < para[2]$ must hold;												
<i>para</i> [2]	window size; $para[1] < para[2] < mb_t$ must hold;												
<i>para</i> [3]	minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; $0 \leq para[3] \leq 100$ must hold;												
<i>para</i> [4]	width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; $0 < para[4] \leq mb_t$ must hold.												
<i>para</i> [5]	the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size; $0 < para[5] \leq para[1]$ must hold.												
n	(global)												

	The order of the globally distributed matrix t . $n \geq 0$.
t	(local) array of size $lld_t * LOC_c(n)$. The local pieces of the global distributed upper quasi-triangular matrix T , in Schur form.
it, jt	(global) The row and column index in the global matrix T indicating the first column of T . $it = jt = 1$ must hold (see Application Notes).
$desct$	(global and local) array of size $dlen_$. The array descriptor for the global distributed matrix T .
q	(local) array of size $lld_q * LOC_c(n)$. On entry, if $compq = 'V'$, the local pieces of the global distributed matrix Q of Schur vectors. If $compq = 'N'$, q is not referenced.
iq, jq	(global) The column index in the global matrix Q indicating the first column of Q . $iq = jq = 1$ must hold (see Application Notes).
$descq$	(global and local) array of size $dlen_$. The array descriptor for the global distributed matrix Q .
$work$	(local workspace) array of size $lwork$
$lwork$	(local) The size of the array $work$. If $lwork = -1$, then a workspace query is assumed; the function only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by pxerbla .
$iwork$	(local workspace) array of size $liwork$
$liwork$	(local) The size of the array $iwork$. If $liwork = -1$, then a workspace query is assumed; the function only calculates the optimal size of the $iwork$ array, returns this value as the first entry of the $iwork$ array, and no error message related to $liwork$ is issued by pxerbla .

OUTPUT Parameters

$select$	(global) array of size n The (partial) reordering is displayed.
t	On exit, t is overwritten by the local pieces of the reordered matrix T , again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks.

q	<p>On exit, if <code>compq = 'V'</code>, q has been postmultiplied by the global orthogonal transformation matrix which reorders t; the leading m columns of q form an orthonormal basis for the specified invariant subspace.</p> <p>If <code>compq = 'N'</code>, q is not referenced.</p>
wr, wi	<p>(global) array of size n</p> <p>The real and imaginary parts, respectively, of the reordered eigenvalues of the matrix T. The eigenvalues are in principle stored in the same order as on the diagonal of T, with $wr[i] = T(i+1,i+1)$ and, if $T(i:i+1,i:i+1)$ is a 2-by-2 diagonal block, $wi[i-1] > 0$ and $wi[i] = -wi[i-1]$.</p> <p>Note also that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.</p>
m	<p>(global)</p> <p>The size of the specified invariant subspace.</p> <p>$0 \leq m \leq n$.</p>
<code>work[0]</code>	On exit, if <code>info = 0</code> , <code>work[0]</code> returns the optimal <code>lwork</code> .
<code>iwork[0]</code>	On exit, if <code>info = 0</code> , <code>iwork[0]</code> returns the optimal <code>liwork</code> .
<code>info</code>	<p>(global)</p> <p>= 0: successful exit</p> <p>< 0: if <code>info = -i</code>, the i-th argument had an illegal value. If the i-th argument is an array and the j-th entry, indexed $j-1$, had an illegal value, then <code>info = -(i*1000+j)</code>, if the i-th argument is a scalar and had an illegal value, then <code>info = -i</code>.</p> <p>> 0: here we have several possibilities</p> <ul style="list-style-type: none"> Reordering of t failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); <p>t may have been partially reordered, and wr and wi contain the eigenvalues in the same order as in t.</p> <p>On exit, <code>info = {the index of t where the swap failed (indexing starts at 1)}</code>.</p> <ul style="list-style-type: none"> A 2-by-2 block to be reordered split into two 1-by-1 blocks and the second block failed to swap with an adjacent block. <p>On exit, <code>info = {the index of t where the swap failed}</code>.</p> <ul style="list-style-type: none"> If <code>info = n+1</code>, there is no valid BLACS context (see the BLACS documentation for details).

Application Notes

The following alignment requirements must hold:

- $mb_t = nb_t = mb_q = nb_q$
- $rsrc_t = rsrc_q$
- $csrc_t = csrc_q$

All matrices must be blocked by a block factor larger than or equal to two (3). This is to simplify reordering across processor borders in the presence of 2-by-2 blocks.

This algorithm cannot work on submatrices of t and q , i.e., $it = jt = iq = jq = 1$ must hold. This is however no limitation since [p?lahqr](#) does not compute Schur forms of submatrices anyway.

Parallel execution recommendations:

- Use a square grid, if possible, for maximum performance. The block parameters in *para* should be kept well below the data distribution block size.
- In general, the parallel algorithm strives to perform as much work as possible without crossing the block borders on the main block diagonal.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trsen

Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.

Syntax

```
void pstrsen(char* job, char* compq, MKL_INT* select, MKL_INT* para, MKL_INT* n, float*
t, MKL_INT* it, MKL_INT* jt, MKL_INT* desct, float* q, MKL_INT* iq, MKL_INT* jq,
MKL_INT* descq, float* wr, float* wi, MKL_INT* m, float* s, float* sep, float* work,
MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);

void pdtrsen(char* job, char* compq, MKL_INT* select, MKL_INT* para, MKL_INT* n, double*
t, MKL_INT* it, MKL_INT* jt, MKL_INT* desct, double* q, MKL_INT* iq, MKL_INT* jq,
MKL_INT* descq, double* wr, double* wi, MKL_INT* m, double* s, double* sep, double*
work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);
```

Include Files

- `mk1_scalapack.h`

Description

`p?trsen` reorders the real Schur factorization of a real matrix $A = Q^*T^*Q^T$, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T , and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace. The reordering is performed by [p?trord](#).

Optionally the function computes the reciprocal condition numbers of the cluster of eigenvalues and/or the invariant subspace.

T must be in Schur form (as returned by [p?lahqr](#)), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

job (global)

Specifies whether condition numbers are required for the cluster of eigenvalues (*s*) or the invariant subspace (*sep*):

= 'N': no condition numbers are required;

= 'E': only the condition number for the cluster of eigenvalues is computed (*s*);

= 'V': only the condition number for the invariant subspace is computed (*sep*);

= 'B': condition numbers for both the cluster and the invariant subspace are computed (*s* and *sep*).

compq

(global)

= 'V': update the matrix *q* of Schur vectors;

= 'N': do not update *q*.

select

(global) array of size *n*

select specifies the eigenvalues in the selected cluster. To select a real eigenvalue *w(j)*, *select[j-1]* must be set to a non-zero number. To select a complex conjugate pair of eigenvalues *w(j)* and *w(j+1)*, corresponding to a 2-by-2 diagonal block, either *select[j-1]* or *select[j]* or both must be set to a non-zero number; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.

para

(global)

Block parameters:

para[0] maximum number of concurrent computational windows allowed in the algorithm; $0 < para[0] \leq \min(NPROW, NPCOL)$ must hold;

para[1] number of eigenvalues in each window; $0 < para[1] < para[2]$ must hold;

para[2] window size; $para[1] < para[2] < mb_t$ must hold;

para[3] minimal percentage of flops required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; $0 \leq para[3] \leq 100$ must hold;

para[4] width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; $0 < para[4] \leq mb_t$ must hold.

para[5] the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size $0 < para[5] \leq para[1]$ must hold;

n

(global)

The order of the globally distributed matrix *t*. $n \geq 0$.

<i>t</i>	(local) array of size $lld_t * LOC_c(n)$. The local pieces of the global distributed upper quasi-triangular matrix <i>T</i> , in Schur form.
<i>it, jt</i>	(global) The row and column index in the global matrix <i>T</i> indicating the first column of <i>T</i> . <i>it</i> = <i>jt</i> = 1 must hold (see Application Notes).
<i>desct</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the global distributed matrix <i>T</i> .
<i>q</i>	(local) array of size $lld_q * LOC_c(n)$. On entry, if <i>compq</i> = 'V', the local pieces of the global distributed matrix <i>Q</i> of Schur vectors. If <i>compq</i> = 'N', <i>q</i> is not referenced.
<i>iq, jq</i>	(global) The column index in the global matrix <i>Q</i> indicating the first column of <i>Q</i> . <i>iq</i> = <i>jq</i> = 1 must hold (see Application Notes).
<i>descq</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the global distributed matrix <i>Q</i> .
<i>work</i>	(local workspace) array of size <i>lwork</i>
<i>lwork</i>	(local) The size of the array <i>work</i> . If <i>lwork</i> = -1, then a workspace query is assumed; the function only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by pxerbla .
<i>iwork</i>	(local workspace) array of size <i>liwork</i>
<i>liwork</i>	(local) The size of the array <i>iwork</i> . If <i>liwork</i> = -1, then a workspace query is assumed; the function only calculates the optimal size of the <i>iwork</i> array, returns this value as the first entry of the <i>iwork</i> array, and no error message related to <i>liwork</i> is issued by pxerbla .

OUTPUT Parameters

<i>t</i>	<i>t</i> is overwritten by the local pieces of the reordered matrix <i>T</i> , again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks.
<i>q</i>	On exit, if <i>compq</i> = 'V', <i>q</i> has been postmultiplied by the global orthogonal transformation matrix which reorders <i>t</i> ; the leading <i>m</i> columns of <i>q</i> form an orthonormal basis for the specified invariant subspace.

If `compq = 'N'`, `q` is not referenced.

`wr, wi`

(global) array of size n

The real and imaginary parts, respectively, of the reordered eigenvalues of the matrix T . The eigenvalues are in principle stored in the same order as on the diagonal of T , with $wr[i] = T(i+1,i+1)$ and, if $T(i:i+1,i:i+1)$ is a 2-by-2 diagonal block, $wi[i-1] > 0$ and $wi[i] = -wi[i-1]$.

Note also that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

`m`

(global)

The size of the specified invariant subspace. $0 \leq m \leq n$.

`s`

(global)

If `job = 'E'` or `'B'`, `s` is a lower bound on the reciprocal condition number for the selected cluster of eigenvalues. `s` cannot underestimate the true reciprocal condition number by more than a factor of \sqrt{n} . If `m = 0` or `n`, `s = 1`.

If `job = 'N'` or `'V'`, `s` is not referenced.

`sep`

(global)

If `job = 'V'` or `'B'`, `sep` is the estimated reciprocal condition number of the specified invariant subspace. If

$m = 0$ or n , `sep = norm(t)`.

If `job = 'N'` or `'E'`, `sep` is not referenced.

`work[0]`

On exit, if `info = 0`, `work[0]` returns the optimal `lwork`.

`iwork[0]`

On exit, if `info = 0`, `iwork[0]` returns the optimal `liwork`.

`info`

(global)

= 0: successful exit

< 0: if `info = -i`, the i -th argument had an illegal value.

If the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value, then `info = -(i*1000+j)`, if the i -th argument is a scalar and had an illegal value, then `info = -i`.

> 0: here we have several possibilities

- Reordering of t failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); t may have been partially reordered, and `wr` and `wi` contain the eigenvalues in the same order as in t .

On exit, `info = {the index of t where the swap failed (indexing starts at 1)}`.

- A 2-by-2 block to be reordered split into two 1-by-1 blocks and the second block failed to swap with an adjacent block.

On exit, `info = {the index of t where the swap failed}`.

- If `info = n+1`, there is no valid BLACS context (see the BLACS documentation for details).

Application Notes

The following alignment requirements must hold:

- $mb_t = nb_t = mb_q = nb_q$
- $rsrc_t = rsrc_q$
- $csrc_t = csrc_q$

All matrices must be blocked by a block factor larger than or equal to two (3). This to simplify reordering across processor borders in the presence of 2-by-2 blocks.

This algorithm cannot work on submatrices of t and q , i.e., $it = jt = iq = jq = 1$ must hold. This is however no limitation since [p?lahqr](#) does not compute Schur forms of submatrices anyway.

For parallel execution, use a square grid, if possible, for maximum performance. The block parameters in *para* should be kept well below the data distribution block size.

In general, the parallel algorithm strives to perform as much work as possible without crossing the block borders on the main block diagonal.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trti2

Computes the inverse of a triangular matrix (local unblocked algorithm).

Syntax

```
void pstrti2 (char *uplo , char *diag , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *info );

void pdtrti2 (char *uplo , char *diag , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *info );

void pctrti2 (char *uplo , char *diag , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *info );

void pztrti2 (char *uplo , char *diag , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?trti2` function computes the inverse of a real/complex upper or lower triangular block matrix sub (A) = $A(ia:ia+n-1, ja:ja+n-1)$.

This matrix should be contained in one and only one process memory space (local operation).

Input Parameters

<i>uplo</i>	(global)
Specifies whether the matrix sub (A) is upper or lower triangular.	
= 'U': sub (A) is upper triangular	
= 'L': sub (A) is lower triangular.	
<i>diag</i>	(global)

Specifies whether or not the matrix A is unit triangular.

= 'N': sub (A) is non-unit triangular

= 'U': sub (A) is unit triangular.

n (global)

The number of rows and columns to be operated on, i.e., the order of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.

a (local)

Pointer into the local memory to an array, size $\text{lld_a} * \text{LOCc}(j_a+n-1)$.

On entry, this array contains the local pieces of the triangular matrix $\text{sub}(A)$.

If $\text{uplo} = 'U'$, the leading n -by- n upper triangular part of the matrix $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced.

If $\text{uplo} = 'L'$, the leading n -by- n lower triangular part of the matrix $\text{sub}(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced. If $\text{diag} = 'U'$, the diagonal elements of $\text{sub}(A)$ are not referenced either and are assumed to be 1.

ia, ja (global)

The row and column indices in the global matrix A indicating the first row and the first column of the $\text{sub}(A)$, respectively.

$desca$ (global and local) array of size $dlen_$. The array descriptor for the distributed matrix A .

Output Parameters

a On exit, the (triangular) inverse of the original matrix, in the same storage format.

$info$

- = 0: successful exit
- < 0: if the i -th argument is an array and the j -th entry, indexed $j-1$, had an illegal value,
then $info = -(i*100+j)$,
- if the i -th argument is a scalar and had an illegal value,
then $info = -i$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?lahqr2

Updates the eigenvalues and Schur decomposition.

Syntax

```
void clahqr2 (const MKL_INT* wantt, const MKL_INT* wantz, const MKL_INT* n, const
MKL_INT* ilo, const MKL_INT* ihi, MKL_Complex8* h, const MKL_INT* ldh, MKL_Complex8* w,
const MKL_INT* iloz, const MKL_INT* ihiz, MKL_Complex8* z, const MKL_INT* ldz, MKL_INT*
info);
```

```
void zlahqr2 (const MKL_INT* wantt, const MKL_INT* wantz, const MKL_INT* n, const
MKL_INT* ilo, const MKL_INT* ihi, MKL_Complex16* h, const MKL_INT* ldh, MKL_Complex16*
w, const MKL_INT* iloz, const MKL_INT* ihiz, MKL_Complex16* z, const MKL_INT* ldz,
MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

?lahqr2 is an auxiliary routine called by ?hseqr to update the eigenvalues and Schur decomposition already computed by ?hseqr, by dealing with the Hessenberg submatrix in rows and columns *ilo* to *ihi*. This version of ?lahqr (not the standard LAPACK version) uses a double-shift algorithm (like LAPACK's ?lahqr). Unlike the standard LAPACK convention, this does not assume the subdiagonal is real, nor does it work to preserve this quality if given.

Input Parameters

<i>wantt</i>	<p>≠ 0: the full Schur form <i>T</i> is required; = 0: only eigenvalues are required.</p>
<i>wantz</i>	<p>≠ 0: the matrix of Schur vectors <i>Z</i> is required; = 0: Schur vectors are not required.</p>
<i>n</i>	The order of the matrix <i>H</i> . <i>n</i> ≥ 0.
<i>ilo, ihi</i>	<p>It is assumed that the matrix <i>H</i> is upper triangular in rows and columns <i>ihi</i> + 1 : <i>n</i>, and that matrix element $H(i_{lo}, i_{lo}-1) = 0$ (unless <i>ilo</i> = 1). ?lahqr works primarily with the Hessenberg submatrix in rows and columns <i>ilo</i> to <i>ihi</i>, but applies transformations to all of <i>h</i> if <i>wantt</i> is nonzero.</p> <p>$1 \leq ilo \leq \max(1, ihi); ihi \leq n.$</p>
<i>h</i>	<p>Array, size <i>ldh</i>*<i>n</i>.</p> <p>On entry, the upper Hessenberg matrix <i>H</i>.</p>
<i>ldh</i>	The leading dimension of the array <i>h</i> . <i>ldh</i> ≥ max(1, <i>n</i>).
<i>iloz, ihiz</i>	<p>Specify the rows of <i>Z</i> to which transformations must be applied if <i>wantz</i> ≠ 0.</p> <p>$1 \leq iloz \leq ilo; ihi \leq ihiz \leq n.$</p>
<i>z</i>	<p>Array, size <i>ldz</i>*<i>n</i>.</p> <p>If <i>wantz</i> ≠ 0, on entry <i>z</i> must contain the current matrix <i>Z</i> of transformations. If <i>wantz</i> = 0, <i>z</i> is not referenced.</p>

ldz The leading dimension of the array *z*. *ldz* $\geq \max(1, n)$.

Output Parameters

<i>h</i>	On exit, if <i>wanttt</i> $\neq 0$, <i>h</i> is upper triangular in rows and columns <i>ilo:ihi</i> . If <i>wanttt</i> = 0, the contents of <i>h</i> are unspecified on exit.
<i>w</i>	Array, size (<i>n</i>) The computed eigenvalues <i>ilo</i> to <i>ihi</i> are stored in the corresponding elements of <i>w</i> . If <i>wanttt</i> $\neq 0$, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in <i>h</i> , with $w[i] = H(i, i)$.
<i>z</i>	If <i>wantz</i> $\neq 0$, on exit <i>z</i> has been updated; transformations are applied only to the submatrix $Z(ilo:ihiz, ilo:ihi)$. If <i>wantz</i> = 0, <i>z</i> is not referenced.
<i>info</i>	= 0: successful exit > 0: if <i>info</i> = <i>i</i> , ?lahqr failed to compute all the eigenvalues <i>ilo</i> to <i>ihi</i> in a total of $30 \cdot (ihi - ilo + 1)$ iterations; elements $w[i:ihi - 1]$ contain those eigenvalues which have been successfully computed.

?lamsh

Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.

Syntax

```
void slamsh (float *s, const MKL_INT *lds, MKL_INT *nbulge, const MKL_INT *jblk, float
*h, const MKL_INT *ldh, const MKL_INT *n, const float *ulp );

void dlamsh (double *s, const MKL_INT *lds, MKL_INT *nbulge, const MKL_INT *jblk,
double *h, const MKL_INT *ldh, const MKL_INT *n, const double *ulp );

void clamsh (MKL_Complex8 *s , const MKL_INT *lds , MKL_INT *nbulge , const MKL_INT
*jblk , MKL_Complex8 *h , const MKL_INT *ldh , const MKL_INT *n , const float *ulp );

void zlamsh (MKL_Complex16 *s , const MKL_INT *lds , MKL_INT *nbulge , const MKL_INT
*jblk , MKL_Complex16 *h , const MKL_INT *ldh , const MKL_INT *n , const double *ulp );
```

Include Files

- mkl_scalapack.h

Description

The ?lamsh function sends multiple shifts through a small (single node) matrix to see how small consecutive subdiagonal elements are modified by subsequent shifts in an effort to maximize the number of bulges that can be sent through. The function should only be called when there are multiple shifts/bulges (*nbulge* > 1) and the first shift is starting in the middle of an unreduced Hessenberg matrix because of two or more small consecutive subdiagonal elements.

Input Parameters

<i>s</i>	(local) Array of size $lds*2*jblk$. On entry, the matrix of shifts. Only the 2x2 diagonal of <i>s</i> is referenced. It is assumed that <i>s</i> has <i>jblk</i> double shifts (size 2).
<i>lds</i>	(local) On entry, the leading dimension of <i>S</i> ; unchanged on exit. $1 < nbulge \leq jblk \leq lds/2$.
<i>nbulge</i>	(local) On entry, the number of bulges to send through <i>h</i> (>1). <i>nbulge</i> should be less than the maximum determined (<i>jblk</i>). $1 < nbulge \leq jblk \leq lds/2$.
<i>jblk</i>	(local) On entry, the number of double shifts determined for <i>S</i> ; unchanged on exit.
<i>h</i>	(local) Array of size $ldh*n$. On entry, the local matrix to apply the shifts on. <i>h</i> should be aligned so that the starting row is 2.
<i>ldh</i>	(local) On entry, the leading dimension of <i>H</i> ; unchanged on exit.
<i>n</i>	(local) On entry, the size of <i>H</i> . If all the bulges are expected to go through, <i>n</i> should be at least $4nbulge+2$. Otherwise, <i>nbulge</i> may be reduced by this function.
<i>ulp</i>	(local) On entry, machine precision. Unchanged on exit.

Output Parameters

<i>s</i>	On exit, the data is rearranged in the best order for applying.
<i>nbulge</i>	On exit, the maximum number of bulges that can be sent through.
<i>h</i>	On exit, the data is destroyed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?lapst

Sorts the numbers in increasing or decreasing order.

Syntax

```
void slapst (const char* id, const MKL_INT* n, const float* d, MKL_INT* indx, MKL_INT* info);
```

```
void dlapst (const char* id, const MKL_INT* n, const double* d, MKL_INT* indx, MKL_INT*
info);
```

Include Files

- mkl_scalapack.h

Description

?lapst is a modified version of the LAPACK routine ?lasrt.

Define a permutation *indx* that sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D').

Use Quick Sort, reverting to Insertion sort on arrays of size ≤ 20 . Dimension of STACK limits *n* to about 2^{32} .

Input Parameters

<i>id</i>	= 'I': sort <i>d</i> in increasing order; = 'D': sort <i>d</i> in decreasing order.
<i>n</i>	The length of the array <i>d</i> .
<i>d</i>	Array, size (<i>n</i>) The array to be sorted.

Output Parameters

<i>indx</i>	Array, size (<i>n</i>). The permutation which sorts the array <i>d</i> .
<i>info</i>	= 0: successful exit < 0: if <i>info</i> = -i, the i-th argument had an illegal value

?laqr6

Performs a single small-bulge multi-shift QR sweep collecting the transformations.

Syntax

```
void slaqr6(char* job, MKL_INT* wantt, MKL_INT* wantz, MKL_INT* kacc22, MKL_INT* n,
MKL_INT* ktop, MKL_INT* kbot, MKL_INT* nshfts, float* sr, float* si, float* h, MKL_INT*
ldh, MKL_INT* iloz, MKL_INT* ihiz, float* z, MKL_INT* ldz, float* v, MKL_INT* ldv,
float* u, MKL_INT* ldu, MKL_INT* nv, float* wv, MKL_INT* ldwv, MKL_INT* nh, float* wh,
MKL_INT* ldwh);
```

```
void dlaqr6(char* job, MKL_INT* wantt, MKL_INT* wantz, MKL_INT* kacc22, MKL_INT* n,
MKL_INT* ktop, MKL_INT* kbot, MKL_INT* nshfts, double* sr, double* si, double* h,
MKL_INT* ldh, MKL_INT* iloz, MKL_INT* ihiz, double* z, MKL_INT* ldz, double* v, MKL_INT*
ldv, double* u, MKL_INT* ldu, MKL_INT* nv, double* wv, MKL_INT* ldwv, MKL_INT* nh,
double* wh, MKL_INT* ldwh);
```

Include Files

- `mkl_scalapack.h`

Description

This auxiliary function performs a single small-bulge multi-shift QR sweep, moving the chain of bulges from top to bottom in the submatrix $H(k_{top}:k_{bot}, k_{top}:k_{bot})$, collecting the transformations in the matrix V or accumulating the transformations in the matrix Z (see below).

This is a modified version of `?laqr5` from LAPACK 3.1.

Input Parameters

<i>job</i>	Set the kind of job to do in <code>?laqr6</code> , as follows: <i>job</i> = 'I': Introduce and chase bulges in submatrix <i>job</i> = 'C': Chase bulges from top to bottom of submatrix <i>job</i> = 'O': Chase bulges off submatrix
<i>wantt</i>	<i>wantt</i> is non-zero if the quasi-triangular Schur factor is being computed. <i>wantt</i> is set to zero otherwise.
<i>wantz</i>	<i>wantz</i> is non-zero if the orthogonal Schur factor is being computed. <i>wantz</i> is set to zero otherwise.
<i>kacc22</i>	Specifies the computation mode of far-from-diagonal orthogonal updates. = 0: <code>?laqr6</code> does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries. = 1: <code>?laqr6</code> accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries. = 2: <code>?laqr6</code> accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.
<i>n</i>	<i>n</i> is the order of the Hessenberg matrix H upon which this function operates.
<i>ktop, kbot</i>	These are the first and last rows and columns of an isolated diagonal block upon which the QR sweep is to be applied. It is assumed without a check that either $k_{top} = 1$ or $H(k_{top}, k_{top}-1) = 0$ and either $k_{bot} = n$ or $H(k_{bot}+1, k_{bot}) = 0$.
<i>nshfts</i>	<i>nshfts</i> gives the number of simultaneous shifts. <i>nshfts</i> must be positive and even.
<i>sr, si</i>	Array of size <i>nshfts</i> <i>sr</i> contains the real parts and <i>si</i> contains the imaginary parts of the <i>nshfts</i> shifts of origin that define the multi-shift QR sweep.
<i>h</i>	Array of size $ldh * n$ On input <i>h</i> contains a Hessenberg matrix H .
<i>ldh</i>	<i>ldh</i> is the leading dimension of H just as declared in the calling function. $ldh \geq \max(1, n)$.

<i>iloz, ihiz</i>	Specify the rows of the matrix <i>Z</i> to which transformations must be applied if <i>wantz</i> is non-zero. $1 \leq iloz \leq ihiz \leq n$
<i>z</i>	<p>Array of size $ldz * ktop$</p> <p>If <i>wantz</i> is non-zero, then the QR sweep orthogonal similarity transformation is accumulated into the matrix $Z(iloz:ihiz, kbot:ktop)$, stored in the array <i>z</i>, from the right.</p> <p>If <i>wantz</i> equals zero, then <i>z</i> is unreferenced.</p>
<i>ldz</i>	<i>ldz</i> is the leading dimension of <i>z</i> just as declared in the calling function. $ldz \geq n$.
<i>v</i>	(workspace) array of size $ldv * nshfts/2$
<i>ldv</i>	<i>ldv</i> is the leading dimension of <i>v</i> as declared in the calling function. $ldv \geq 3$.
<i>u</i>	(workspace) array of size $ldu * (3*nshfts-3)$
<i>ldu</i>	<i>ldu</i> is the leading dimension of <i>u</i> just as declared in the calling function. $ldu \geq 3*nshfts-3$.
<i>nh</i>	<i>nh</i> is the number of columns in array <i>wh</i> available for workspace. $nh \geq 1$ is required for usage of this workspace, otherwise the updates of the far-from-diagonal elements will be updated without level 3 BLAS.
<i>wh</i>	(workspace) array of size $ldwh * nh$
<i>ldwh</i>	Leading dimension of <i>wh</i> just as declared in the calling function. $ldwh \geq 3*nshfts-3$.
<i>nv</i>	<i>nv</i> is the number of rows in <i>wv</i> available for workspace. $nv \geq 1$ is required for usage of this workspace, otherwise the updates of the far-from-diagonal elements will be updated without level 3 BLAS.
<i>wv</i>	(workspace) array of size $ldwv * 3*nshfts$
<i>ldwv</i>	<p>scalar</p> <p><i>ldwv</i> is the leading dimension of <i>wv</i> as declared in the in the calling function. $ldwv \geq nv$.</p>

OUTPUT Parameters

<i>h</i>	A multi-shift QR sweep with shifts $sr[j] + i*si[j]$ is applied to the isolated diagonal block in matrix rows and columns <i>ktop</i> through <i>kbot</i> .
<i>z</i>	<p>If <i>wantz</i> is non-zero, then the QR sweep orthogonal/unitary similarity transformation is accumulated into the matrix $Z(iloz:ihiz, kbot:ktop)$ from the right.</p> <p>If <i>wantz</i> equals zero, then <i>z</i> is unreferenced.</p>

Application Notes

Notes

Based on contributions by Karen Braman and Ralph Byers, Department of Mathematics, University of Kansas, USA Robert Granat, Department of Computing Science and HPC2N, Umea University, Sweden

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?slar1va

Computes scaled eigenvector corresponding to given eigenvalue.

Syntax

```
void slar1va(MKL_INT* n, MKL_INT* b1, MKL_INT* bn, float* lambda, float* d, float* l,
float* ld, float* lld, float* pivmin, float* gaptol, float* z, MKL_INT* wantnc, MKL_INT*
negcnt, float* ztz, float* mingma, MKL_INT* r, MKL_INT* isuppz, float* nrminv, float*
resid, float* rqcorr, float* work);
```

```
void dlar1va(MKL_INT* n, MKL_INT* b1, MKL_INT* bn, double* lambda, double* d, double* l,
double* ld, double* lld, double* pivmin, double* gaptol, double* z, MKL_INT* wantnc,
MKL_INT* negcnt, double* ztz, double* mingma, MKL_INT* r, MKL_INT* isuppz, double*
nrminv, double* resid, double* rqcorr, double* work);
```

Include Files

- mkl_scalapack.h

Description

?slar1va computes the (scaled) r -th column of the inverse of the submatrix in rows $b1$ through bn of the tridiagonal matrix $LDL^T - \lambda I$. When λ is close to an eigenvalue, the computed vector is an accurate eigenvector. Usually, r corresponds to the index where the eigenvector is largest in magnitude. The following steps accomplish this computation :

1. Stationary qd transform, $LDL^T - \lambda I = L_+ D_+ L_+^T$,
2. Progressive qd transform, $LDL^T - \lambda I = U_- D_- U_-^T$,
3. Computation of the diagonal elements of the inverse of $LDL^T - \lambda I$ by combining the above transforms, and choosing r as the index where the diagonal of the inverse is (one of the) largest in magnitude.
4. Computation of the (scaled) r -th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

n	The order of the matrix LDL^T .
$b1$	First index of the submatrix of LDL^T .
bn	Last index of the submatrix of LDL^T .
$lambda$	The shift λ . In order to compute an accurate eigenvector, $lambda$ should be a good approximation to an eigenvalue of LDL^T .
l	Array of size $n-1$ The $(n-1)$ subdiagonal elements of the unit bidiagonal matrix L , in elements 0 to $n-2$.
d	Array of size n The n diagonal elements of the diagonal matrix D .
ld	Array of size $n-1$ The $n-1$ elements $l[i]*d[i]$, $i=0,\dots,n-2$.

<i>lld</i>	Array of size $n-1$ The $n-1$ elements $l[i]*l[i]*d[i], i=0,\dots,n-2$.
<i>pivmin</i>	The minimum pivot in the Sturm sequence.
<i>gaptol</i>	Tolerance that indicates when eigenvector entries are negligible with respect to their contribution to the residual.
<i>z</i>	Array of size n On input, all entries of z must be set to 0.
<i>wantnc</i>	Specifies whether <i>negcnt</i> has to be computed.
<i>r</i>	The twist index for the twisted factorization used to compute z . On input, $0 \leq r \leq n$. If r is input as 0, r is set to the index where $(LDL^T - \sigma I)^{-1}$ is largest in magnitude. If $1 \leq r \leq n$, r is unchanged. Ideally, r designates the position of the maximum entry in the eigenvector.
<i>work</i>	(Workspace) array of size $4*n$

OUTPUT Parameters

<i>z</i>	On output, z contains the (scaled) r -th column of the inverse. The scaling is such that $z[r-1]$ equals 1.
<i>negcnt</i>	If <i>wantnc</i> is non-zero then <i>negcnt</i> = the number of pivots $< pivmin$ in the matrix factorization LDL^T , and <i>negcnt</i> = -1 otherwise.
<i>ztz</i>	The square of the 2-norm of z .
<i>mingma</i>	The reciprocal of the largest (in magnitude) diagonal element of the inverse of $LDL^T - \sigma I$.
<i>r</i>	On output, r contains the twist index used to compute z .
<i>isuppz</i>	array of size 2 The support of the vector in z , i.e., the vector z is non-zero only in elements <i>isuppz</i> [0] and <i>isuppz</i> [1].
<i>nrminv</i>	$nrminv = 1/\text{SQRT}(ztz)$
<i>resid</i>	The residual of the FP vector. $resid = \text{ABS}(mingma)/\text{SQRT}(ztz)$
<i>rqcorr</i>	The Rayleigh Quotient correction to $lambda$.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?leref

Applies Householder reflectors to matrices on their rows or columns.

Syntax

```
void slaref (const char* type, float* a, const MKL_INT* lda, const MKL_INT* wantz,
float* z, const MKL_INT* ldz, const MKL_INT* block, MKL_INT* irow1, MKL_INT* icoll,
const MKL_INT* istart, const MKL_INT* istop, const MKL_INT* itmp1, const MKL_INT*
itmp2, const MKL_INT* liloz, const MKL_INT* li hiz, const float* vecs, float* v2, float*
v3, float* t1, float* t2, float* t3);
```

```
void dlaref (const char* type, double* a, const MKL_INT* lda, const MKL_INT* wantz,
double* z, const MKL_INT* ldz, const MKL_INT* block, MKL_INT* irow1, MKL_INT* icoll,
const MKL_INT* istart, const MKL_INT* istop, const MKL_INT* itmp1, const MKL_INT*
itmp2, const MKL_INT* liloz, const MKL_INT* li hiz, const double* vecs, double* v2,
double* v3, double* t1, double* t2, double* t3);
```

```
void claref (const char* type, MKL_Complex8* a, const MKL_INT* lda, const MKL_INT*
wantz, MKL_Complex8* z, const MKL_INT* ldz, const MKL_INT* block, MKL_INT* irow1,
MKL_INT* icoll, const MKL_INT* istart, const MKL_INT* istop, const MKL_INT* itmp1,
const MKL_INT* itmp2, const MKL_INT* liloz, const MKL_INT* li hiz, const MKL_Complex8*
vecs, MKL_Complex8* v2, MKL_Complex8* v3, MKL_Complex8* t1, MKL_Complex8* t2,
MKL_Complex8* t3);
```

```
void zlaref (const char* type, MKL_Complex16* a, const MKL_INT* lda, const MKL_INT*
wantz, MKL_Complex16* z, const MKL_INT* ldz, const MKL_INT* block, MKL_INT* irow1,
MKL_INT* icoll, const MKL_INT* istart, const MKL_INT* istop, const MKL_INT* itmp1,
const MKL_INT* itmp2, const MKL_INT* liloz, const MKL_INT* li hiz, const MKL_Complex16*
vecs, MKL_Complex16* v2, MKL_Complex16* v3, MKL_Complex16* t1, MKL_Complex16* t2,
MKL_Complex16* t3);
```

Include Files

- mkl_scalapack.h

Description

?laref applies one or several Householder reflectors of size 3 to one or two matrices (if column is specified) on either their rows or columns.

Input Parameters

<i>type</i>	(local) If 'R': Apply reflectors to the rows of the matrix (apply from left) Otherwise: Apply reflectors to the columns of the matrix Unchanged on exit.
<i>a</i>	(local) Array, $lld_a * LOCc(ja + n - 1)$ On entry, the matrix to receive the reflections.
<i>lda</i>	(local) On entry, the leading dimension of <i>a</i> . Unchanged on exit.
<i>wantz</i>	(local)

If $wantz \neq 0$, then apply any column reflections to z as well.

If $wantz = 0$, then do no additional work on z .

z	<p>(local)</p> <p>Array, $ldz * ncols$, where the value $ncols$ depends on other arguments. If $wantz \neq 0$ and $type \neq 'R'$ then $ncols = icol1 + 3 * (lihi - lilo + 1)$. Otherwise, $ncols$ is unused.</p> <p>On entry, the second matrix to receive column reflections.</p> <p>This is changed only if $wantz$ is set.</p>
ldz	<p>(local)</p> <p>On entry, the leading dimension of z.</p> <p>Unchanged on exit.</p>
$block$	<p>(local)</p> <p>If nonzero, then apply several reflectors at once and read their data from the $vecs$ array.</p> <p>If zero, apply the single reflector given by $v2$, $v3$, $t1$, $t2$, and $t3$.</p>
$irow1$	<p>(local)</p> <p>On entry, the local row element of a.</p>
$icol1$	<p>(local)</p> <p>On entry, the local column element of a.</p>
$istart$	<p>(local)</p> <p>Specifies the "number" of the first reflector. This is used as an index into $vecs$ if $block$ is set. $istart$ is ignored if $block$ is zero.</p>
$istop$	<p>(local)</p> <p>Specifies the "number" of the last reflector. This is used as an index into $vecs$ if $block$ is set. $istop$ is ignored if $block$ is zero.</p>
$itmp1$	<p>(local)</p> <p>Starting range into a. For rows, this is the local first column. For columns, this is the local first row.</p>
$itmp2$	<p>(local)</p> <p>Ending range into a. For rows, this is the local last column. For columns, this is the local last row.</p>
$lilo, lihi$	<p>(local)</p> <p>These serve the same purpose as $itmp1$, $itmp2$ but for z when $wantz$ is set.</p>
$vecs$	<p>(local)</p>

Array of size 3*N (matrix size)

This holds the size 3 reflectors one after another and this is only accessed when *block* is nonzero

v2, v3, t1, t2, t3

(local)

This holds information on a single size 3 Householder reflector and is read when *block* is zero, and overwritten when *block* is nonzero

Output Parameters

a

The updated matrix on exit.

z

This is changed only if *wantz* is set.

irow1

Undefined on output.

icol1

Undefined on output.

v2, v3, t1, t2, t3

Overwritten when *block* is nonzero.

?larrb2

Provides limited bisection to locate eigenvalues for more accuracy.

Syntax

```
void slarrb2(MKL_INT* n, float* d, float* lld, MKL_INT* ifirst, MKL_INT* ilast, float*
rtol1, float* rtol2, MKL_INT* offset, float* w, float* wgap, float* werr, float* work,
MKL_INT* iwork, float* pivmin, float* lgpvmn, float* lgspdm, MKL_INT* twist, MKL_INT*
info);
```

```
void dlarrb2(MKL_INT* n, double* d, double* lld, MKL_INT* ifirst, MKL_INT* ilast,
double* rtol1, double* rtol2, MKL_INT* offset, double* w, double* wgap, double* werr,
double* work, MKL_INT* iwork, double* pivmin, double* lgpvmn, double* lgspdm, MKL_INT*
twist, MKL_INT* info);
```

Include Files

- `mk1_scalapack.h`

Description

Given the relatively robust representation (RRR) LDL^T , ?larrb2 does "limited" bisection to refine the eigenvalues of LDL^T with indices in a given range to more accuracy. Initial guesses for these eigenvalues are input in *w*, the corresponding estimate of the error in these guesses and their gaps are input in *werr* and *wgap*, respectively. During bisection, intervals [*left*, *right*] are maintained by storing their mid-points and semi-widths in the arrays *w* and *werr* respectively. The range of indices is specified by the *ifirst*, *ilast*, and *offset* parameters, as explained in [Input Parameters](#).

NOTE

There are very few minor differences between `larrrb` from LAPACK and this current function `?larrrb2`. The most important reason for creating this nearly identical copy is profiling: in the ScaLAPACK MRRR algorithm, eigenvalue computation using `?larrrb2` is used for refinement in the construction of the representation tree, as opposed to the initial computation of the eigenvalues for the root RRR which uses `?larrrb`. When profiling, this allows an easy quantification of refinement work vs. computing eigenvalues of the root.

Input Parameters

<code>n</code>	The order of the matrix.
<code>d</code>	Array of size <code>n</code> . The <code>n</code> diagonal elements of the diagonal matrix D .
<code>lld</code>	Array of size <code>n-1</code> . The $(n-1)$ elements $l_{i+1} * l_{i+1} * d[i]$, $i=0, \dots, n-2$.
<code>ifirst</code>	The index of the first eigenvalue to be computed.
<code>ilast</code>	The index of the last eigenvalue to be computed.
<code>rtol1, rtol2</code>	Tolerance for the convergence of the bisection intervals. An interval $[left, right]$ has converged if $right - left < \max(rtol1 * gap, rtol2 * \max(left , right))$ where gap is the (estimated) distance to the nearest eigenvalue.
<code>offset</code>	Offset for the arrays <code>w</code> , <code>wgap</code> and <code>werr</code> , i.e., the elements indexed <code>ifirst - offset - 1</code> through <code>ilast - offset - 1</code> of these arrays are to be used.
<code>w</code>	Array of size <code>n</code> On input, <code>w[ifirst - offset - 1]</code> through <code>w[ilast - offset - 1]</code> are estimates of the eigenvalues of LDL^T indexed <code>ifirst</code> through <code>ilast</code> .
<code>wgap</code>	Array of size <code>n-1</code> . On input, the (estimated) gaps between consecutive eigenvalues of LDL^T , i.e., <code>wgap[I - offset - 1]</code> is the gap between eigenvalues I and $I + 1$. Note that if <code>ifirst = ilast</code> then <code>wgap[ifirst - offset - 1]</code> must be set to zero.
<code>werr</code>	Array of size <code>n</code> . On input, <code>werr[ifirst - offset - 1]</code> through <code>werr[ilast - offset - 1]</code> are the errors in the estimates of the corresponding elements in <code>w</code> .
<code>work</code>	(workspace) array of size $4 * n$. Workspace.
<code>iwork</code>	(workspace) array of size $2 * n$. Workspace.
<code>pivmin</code>	The minimum pivot in the Sturm sequence.

<i>lgpvmn</i>	Logarithm of <i>pivmin</i> , precomputed.
<i>lgspdm</i>	Logarithm of the spectral diameter, precomputed.
<i>twist</i>	The twist index for the twisted factorization that is used for the <i>negcount</i> . $twist = n$: Compute <i>negcount</i> from $LDL^T - \lambda I = L_+ D_+ L_+^T$ $twist = 1$: Compute <i>negcount</i> from $LDL^T - \lambda I = U \cdot D \cdot U^T$ $twist = r, 1 < r < n$: Compute <i>negcount</i> from $LDL^T - \lambda I = N_r \Delta_r N_r^T$

OUTPUT Parameters

<i>w</i>	On output, the eigenvalue estimates in <i>w</i> are refined.
<i>wgap</i>	On output, the eigenvalue gaps in <i>wgap</i> are refined.
<i>werr</i>	On output, the errors in <i>werr</i> are refined.
<i>info</i>	Error flag.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larrrd2

Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Syntax

```
void slarrrd2(char* range, char* order, MKL_INT* n, float* vl, float* vu, MKL_INT* il,
MKL_INT* iu, float* gers, float* reltol, float* d, float* e, float* e2, float* pivmin,
MKL_INT* nsplit, MKL_INT* isplit, MKL_INT* m, float* w, float* werr, float* wl, float*
wu, MKL_INT* iblock, MKL_INT* indexw, float* work, MKL_INT* iwork, MKL_INT* dol,
MKL_INT* dou, MKL_INT* info);

void dlarrrd2(char* range, char* order, MKL_INT* n, double* vl, double* vu, MKL_INT* il,
MKL_INT* iu, double* gers, double* reltol, double* d, double* e, double* e2, double*
pivmin, MKL_INT* nsplit, MKL_INT* isplit, MKL_INT* m, double* w, double* werr, double*
wl, double* wu, MKL_INT* iblock, MKL_INT* indexw, double* work, MKL_INT* iwork, MKL_INT*
dol, MKL_INT* dou, MKL_INT* info);
```

Include Files

- `mkl_scalapack.h`

Description

`?larrrd2` computes the eigenvalues of a symmetric tridiagonal matrix *T* to limited initial accuracy. This is an auxiliary code to be called from [larre2a](#).

`?larrrd2` has been created using the LAPACK code `larrrd` which itself stems from [stebz](#). The motivation for creating `?larrrd2` is efficiency: When computing eigenvalues in parallel and the input tridiagonal matrix splits into blocks, `?larrrd2` can skip over blocks which contain none of the eigenvalues from DOL to DOU for which the processor responsible. In extreme cases (such as large matrices consisting of many blocks of small size like 2x2), the gain can be substantial.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>range</i>	<p>= 'A': ("All") all eigenvalues will be found.</p> <p>= 'V': ("Value") all eigenvalues in the half-open interval (v_l, v_u] will be found.</p> <p>= 'I': ("Index") eigenvalues of the entire matrix with the indices in a given range will be found.</p>
<i>order</i>	<p>= 'B': ("By Block") the eigenvalues will be grouped by split-off block (see <i>iblock</i>, <i>isplit</i>) and ordered from smallest to largest within the block.</p> <p>= 'E': ("Entire matrix") the eigenvalues for the entire matrix will be ordered from smallest to largest.</p>
<i>n</i>	The order of the tridiagonal matrix T . $n \geq 0$.
<i>v_l</i> , <i>v_u</i>	<p>If <i>range</i>='V', the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to v_l, or greater than v_u, will not be returned. $v_l < v_u$.</p> <p>Not referenced if <i>range</i> = 'A' or 'I'.</p>
<i>il</i> , <i>iu</i>	<p>If <i>range</i>='I', the indices (in ascending order) of the smallest eigenvalue, to be returned in $w[il-1]$, and largest eigenvalue, to be returned in $w[iu-1]$.</p> <p>$1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$ if $n = 0$.</p> <p>Not referenced if <i>range</i> = 'A' or 'V'.</p>
<i>gers</i>	<p>Array of size $2*n$</p> <p>The n Gerschgorin intervals (the i-th Gerschgorin interval is ($gers[2*i-2]$, $gers[2*i-1]$)).</p>
<i>reltol</i>	The minimum relative width of an interval. When an interval is narrower than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. Note: this should always be at least $\text{radix} * \text{machine epsilon}$.
<i>d</i>	<p>Array of size n</p> <p>The n diagonal elements of the tridiagonal matrix T.</p>
<i>e</i>	<p>Array of size $n-1$</p> <p>The $(n-1)$ off-diagonal elements of the tridiagonal matrix T.</p>
<i>e2</i>	<p>Array of size $n-1$</p> <p>The $(n-1)$ squared off-diagonal elements of the tridiagonal matrix T.</p>
<i>pivmin</i>	The minimum pivot allowed in the sturm sequence for T .

<i>nsplit</i>	The number of diagonal blocks in the matrix T . $1 \leq nsplit \leq n$.
<i>isplit</i>	Array of size n The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to <i>isplit</i> [0], the second of rows/columns <i>isplit</i> [0]+1 through <i>isplit</i> [1], etc., and the <i>nsplit</i> -th submatrix consists of rows/columns <i>isplit</i> [<i>nsplit</i> -2]+1 through <i>isplit</i> [<i>nsplit</i> -1]= n . (Only the first <i>nsplit</i> elements will actually be used, but since the user cannot know a priori what value <i>nsplit</i> will have, n words must be reserved for <i>isplit</i> .)
<i>work</i>	(workspace) Array of size $4*n$
<i>iwork</i>	(workspace) Array of size $3*n$
<i>dol, dou</i>	Specifying an index range <i>dol</i> : <i>dou</i> allows the user to work on only a selected part of the representation tree. Otherwise, the setting <i>dol</i> =1, <i>dou</i> = n should be applied. Note that <i>dol</i> and <i>dou</i> refer to the order in which the eigenvalues are stored in W .

OUTPUT Parameters

<i>m</i>	The actual number of eigenvalues found. $0 \leq m \leq n$. (See also the description of <i>info</i> =2,3.)
<i>w</i>	Array of size n On exit, the first m elements of w will contain the eigenvalue approximations. ?larnd2 computes an interval $I_j = (a_j, b_j]$ that includes eigenvalue j . The eigenvalue approximation is given as the interval midpoint $w[j-1] = (a_j + b_j)/2$. The corresponding error is bounded by $werr[j-1] = \text{abs}(a_j - b_j)/2$.
<i>werr</i>	Array of size n The error bound on the corresponding eigenvalue approximation in w .
<i>wl, wu</i>	The interval (wl, wu] contains all the wanted eigenvalues. If <i>range</i> ='V', then $wl=v_l$ and $wu=v_u$. If <i>range</i> ='A', then wl and wu are the global Gerschgorin bounds on the spectrum. If <i>range</i> ='I', then wl and wu are computed by SLAEBZ from the index range specified.
<i>iblock</i>	Array of size n

At each row/column j where $e[j-1]$ is zero or small, the matrix T is considered to split into a block diagonal matrix. On exit, if $info = 0$, $iblock[i]$ specifies to which block (from 0 to the number of blocks minus one) the eigenvalue $w[i]$ belongs. (`?larre2` may use the remaining $n-m$ elements as workspace.)

indexw

Array of size n

The indices of the eigenvalues within each block (submatrix); for example, $indexw[i]=j$ and $iblock[i]=k$ imply that the $(i+1)$ -th eigenvalue $w[i]$ is the j -th eigenvalue in block k .

info

= 0: successful exit

< 0: if $info = -i$, the i -th argument had an illegal value

> 0: some or all of the eigenvalues failed to converge or were not computed:

- =1 or 3: Bisection failed to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.
- =2 or 3: $range='I'$ only: Not all of the eigenvalues $il:iu$ were found.
- = 4: $range='I'$, and the Gershgorin interval initially used was too small. No eigenvalues were computed.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larre2

Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.

Syntax

```
void slarre2(char* range, MKL_INT* n, float* vl, float* vu, MKL_INT* il, MKL_INT* iu,
float* d, float* e, float* e2, float* rtol1, float* rtol2, float* spltol, MKL_INT*
nsplit, MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou, float* w, float* werr,
float* wgap, MKL_INT* iblock, MKL_INT* indexw, float* gers, float* pivmin, float* work,
MKL_INT* iwork, MKL_INT* info);
```

```
void dlarre2(char* range, MKL_INT* n, double* vl, double* vu, MKL_INT* il, MKL_INT* iu,
double* d, double* e, double* e2, double* rtol1, double* rtol2, double* spltol, MKL_INT*
nsplit, MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou, double* w, double*
werr, double* wgap, MKL_INT* iblock, MKL_INT* indexw, double* gers, double* pivmin,
double* work, MKL_INT* iwork, MKL_INT* info);
```

Include Files

- `mkc_scalapack.h`

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T , `?larre2` sets, via `?larra`, "small" off-diagonal elements to zero. For each block T_i , it finds

- a suitable shift at one end of the block's spectrum,

- the root RRR, $T_i - \sigma_i I = L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

The representations and eigenvalues found are then returned to `?stegr2` to compute the eigenvectors T .

`?larre2` is more suitable for parallel computation than the original LAPACK code for computing the root RRR and its eigenvalues. When computing eigenvalues in parallel and the input tridiagonal matrix splits into blocks, `?larre2` can skip over blocks which contain none of the eigenvalues from d_{ol} to d_{ou} for which the processor is responsible. In extreme cases (such as large matrices consisting of many blocks of small size, e.g. 2x2), the gain can be substantial.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>range</i>	<p>= 'A': ("All") all eigenvalues will be found.</p> <p>= 'V': ("Value") all eigenvalues in the half-open interval $(vl, vu]$ will be found.</p> <p>= 'I': ("Index") eigenvalues of the entire matrix with the indices in a given range will be found.</p>
<i>n</i>	The order of the matrix. $n > 0$.
<i>vl, vu</i>	<p>If <i>range</i>='V', the lower and upper bounds for the eigenvalues.</p> <p>Eigenvalues less than or equal to <i>vl</i>, or greater than <i>vu</i>, will not be returned. $vl < vu$.</p>
<i>il, iu</i>	<p>If <i>range</i>='I', the indices (in ascending order) of the smallest eigenvalue, to be returned in $w[il-1]$, and largest eigenvalue, to be returned in $w[iu-1]$.</p> <p>$1 \leq il \leq iu \leq n$.</p>
<i>d</i>	<p>Array of size n</p> <p>The n diagonal elements of the tridiagonal matrix T.</p>
<i>e</i>	<p>Array of size n</p> <p>The first $(n-1)$ entries contain the subdiagonal elements of the tridiagonal matrix T; $e[n-1]$ need not be set.</p>
<i>e2</i>	<p>Array of size n</p> <p>The first $(n-1)$ entries contain the squares of the subdiagonal elements of the tridiagonal matrix T; $e2[n-1]$ need not be set.</p>
<i>rtol1, rtol2</i>	<p>Parameters for bisection.</p> <p>An interval $[left, right]$ has converged if $right-left < \max(rtol1*gap, rtol2*\max(left , right))$</p>
<i>spltol</i>	The threshold for splitting.

dol, dou Specifying an index range *dol:dou* allows the user to work on only a selected part of the representation tree. Otherwise, the setting *dol=1, dou=n* should be applied.

Note that *dol* and *dou* refer to the order in which the eigenvalues are stored in *w*.

work Workspace array of size $6*n$

iwork Workspace array of size $5*n$

OUTPUT Parameters

vl, vu If *range*='I' or ='A', *?larre2* contains bounds on the desired part of the spectrum.

d The n diagonal elements of the diagonal matrices D_i .

e *e* contains the subdiagonal elements of the unit bidiagonal matrices L_i . The entries $e[isplit[i]]$, $0 \leq i < nsplit$, contain the base points σ_{i+1} on output.

e2 The entries $e2[isplit[i]]$, $0 \leq i < nsplit$, are set to zero.

nsplit The number of blocks *T* splits into. $1 \leq nsplit \leq n$.

isplit Array of size n

The splitting points, at which *T* breaks up into blocks.

The first block consists of rows/columns 1 to $isplit[0]$, the second of rows/columns $isplit[0]+1$ through $isplit[1]$, etc., and the *nsplit*-th block consists of rows/columns $isplit[nsplit-2]+1$ through $isplit[nsplit-1]=n$.

m The total number of eigenvalues (of all $L_i D_i L_i^T$) found.

w Array of size n

The first m elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i D_i L_i^T$, are sorted in ascending order (*?larre2* may use the remaining $n-m$ elements as workspace).

Note that immediately after exiting this function, only the eigenvalues in *w* with indices in range *dol-1:dou-1* might rely on this processor when the eigenvalue computation is done in parallel.

werr Array of size n

The error bound on the corresponding eigenvalue in *w*.

Note that immediately after exiting this function, only the uncertainties in *werr* with indices in range *dol-1:dou-1* might rely on this processor when the eigenvalue computation is done in parallel.

wgap Array of size n

The separation from the right neighbor eigenvalue in *w*.

The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree.

Exception: at the right end of a block we store the left gap

Note that immediately after exiting this function, only the gaps in *wgap* with indices in range *dol*-1:*dou*-1 might rely on this processor when the eigenvalue computation is done in parallel.

iblock

Array of size *n*

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in *w*; *iblock*[*i*]=1 if eigenvalue *w*[*i*] belongs to the first block from the top, *iblock*[*i*]=2 if *w*[*i*] belongs to the second block, and so on.

indexw

Array of size *n*

The indices of the eigenvalues within each block (submatrix); for example, *indexw*[*i*]= 10 and *iblock*[*i*]=2 imply that the (*i*+1)-th eigenvalue *w*[*i*] is the 10th eigenvalue in block 2.

gers

Array of size $2*n$

The *n* Gerschgorin intervals (the *i*-th Gerschgorin interval is (*gers*[$2*i-2$], *gers*[$2*i-1$])).

pivmin

The minimum pivot in the sturm sequence for *T*.

info

= 0: successful exit

> 0: A problem occurred in ?larre2.

< 0: One of the called functions signaled an internal problem.

Needs inspection of the corresponding parameter *info* for further information.

=-1: Problem in ?larrd.

=-2: Not enough internal iterations to find the base representation.

=-3: Problem in ?larrb when computing the refined root representation for ?lasq2.

=-4: Problem in ?larrb when performing bisection on the desired part of the spectrum.

=-5: Problem in ?lasq2

=-6: Problem in ?lasq2

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larre2a

Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.

Syntax

```
void slarre2a(char* range, MKL_INT* n, float* vl, float* vu, MKL_INT* il, MKL_INT* iu,
float* d, float* e, float* e2, float* rtol1, float* rtol2, float* spltol, MKL_INT*
nsplit, MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou, MKL_INT* needil,
```

```

MKL_INT* neediu, float* w, float* werr, float* wgap, MKL_INT* iblock, MKL_INT* indexw,
float* gers, float* sdiam, float* pivmin, float* work, MKL_INT* iwork, float* minrgp,
MKL_INT* info);

void dlarre2a(char* range, MKL_INT* n, double* vl, double* vu, MKL_INT* il, MKL_INT* iu,
double* d, double* e, double* e2, double* rtol1, double* rtol2, double* spltol, MKL_INT*
nsplit, MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou, MKL_INT* needil,
MKL_INT* neediu, double* w, double* werr, double* wgap, MKL_INT* iblock, MKL_INT*
indexw, double* gers, double* sdiam, double* pivmin, double* work, MKL_INT* iwork,
double* minrgp, MKL_INT* info);

```

Include Files

- mkl_scalapack.h

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T , `?larre2a` sets any "small" off-diagonal elements to zero, and for each unreduced block T_i , it finds

- a suitable shift at one end of the block's spectrum,
- the base representation, $T_i - \sigma_i I = L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

NOTE

The algorithm obtains a crude picture of all the wanted eigenvalues (as selected by `range`). However, to reduce work and improve scalability, only the eigenvalues `dol` to `dou` are refined. Furthermore, if the matrix splits into blocks, RRRs for blocks that do not contain eigenvalues from `dol` to `dou` are skipped. The DQDS algorithm (function `?lasq2`) is not used, unlike in the sequential case. Instead, eigenvalues are computed in parallel to some figures using bisection.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<code>range</code>	<p>= 'A': ("All") all eigenvalues will be found.</p> <p>= 'V': ("Value") all eigenvalues in the half-open interval $(vl, vu]$ will be found.</p> <p>= 'I': ("Index") eigenvalues of the entire matrix with the indices in a given range will be found.</p>
<code>n</code>	The order of the matrix. $n > 0$.
<code>vl, vu</code>	<p>If <code>range='V'</code>, the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to <code>vl</code>, or greater than <code>vu</code>, will not be returned. $vl < vu$.</p> <p>If <code>range='I'</code> or <code>= 'A'</code>, <code>?larre2a</code> computes bounds on the desired part of the spectrum.</p>

<i>il, iu</i>	<p>If <i>range</i>='I', the indices (in ascending order) of the smallest eigenvalue, to be returned in <i>w</i>[<i>il</i>-1], and largest eigenvalue, to be returned in <i>w</i>[<i>iu</i>-1].</p> <p>$1 \leq il \leq iu \leq n$.</p>
<i>d</i>	<p>Array of size <i>n</i></p> <p>On entry, the <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i>.</p>
<i>e</i>	<p>Array of size <i>n</i></p> <p>The first (<i>n</i>-1) entries contain the subdiagonal elements of the tridiagonal matrix <i>T</i>; <i>e</i>[<i>n</i>-1] need not be set.</p>
<i>e2</i>	<p>Array of size <i>n</i></p> <p>The first (<i>n</i>-1) entries contain the squares of the subdiagonal elements of the tridiagonal matrix <i>T</i>; <i>e2</i>[<i>n</i>-1] need not be set.</p>
<i>rtol1, rtol2</i>	<p>Parameters for bisection.</p> <p>An interval [<i>left</i>,<i>right</i>] has converged if <i>right</i> - <i>left</i> < max(<i>rtol1</i>*<i>gap</i>, <i>rtol2</i>*max(<i>left</i> , <i>right</i>))</p>
<i>spltol</i>	The threshold for splitting.
<i>dol, dou</i>	<p>If the user wants to work on only a selected part of the representation tree, he can specify an index range <i>dol</i>:<i>dou</i>.</p> <p>Otherwise, the setting <i>dol</i>=1, <i>dou</i>=<i>n</i> should be applied.</p> <p>Note that <i>dol</i> and <i>dou</i> refer to the order in which the eigenvalues are stored in <i>w</i>.</p>
<i>work</i>	Workspace array of size 6* <i>n</i>
<i>iwork</i>	Workspace array of size 5* <i>n</i>
<i>minrgp</i>	The minimum relative gap threshold to decide whether an eigenvalue or a cluster boundary is reached.

OUTPUT Parameters

<i>vl, vu</i>	<p>If <i>range</i>='V', the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to <i>vl</i>, or greater than <i>vu</i>, are not returned. <i>vl</i> < <i>vu</i>.</p> <p>If <i>range</i>='I' or <i>range</i>='A', ?larre2a computes bounds on the desired part of the spectrum.</p>
<i>d</i>	The <i>n</i> diagonal elements of the diagonal matrices <i>D_i</i> .
<i>e</i>	<i>e</i> contains the subdiagonal elements of the unit bidiagonal matrices <i>L_i</i> . The entries <i>e</i> [<i>isplit</i> [<i>i</i>]], 0 ≤ <i>i</i> < <i>nsplit</i> , contain the base points σ_{i+1} on output.
<i>e2</i>	The entries <i>e2</i> [<i>isplit</i> [<i>i</i>]], 0 ≤ <i>i</i> < <i>nsplit</i> have been set to zero.
<i>nsplit</i>	The number of blocks <i>T</i> splits into. $1 \leq nsplit \leq n$.
<i>isplit</i>	<p>Array of size <i>n</i></p> <p>The splitting points, at which <i>T</i> breaks up into blocks.</p>

The first block consists of rows/columns 1 to $isplit[0]$, the second of rows/columns $isplit[0]+1$ through $isplit[1]$, etc., and the $nsplit$ -th block consists of rows/columns $isplit[nsplit-2]+1$ through $isplit[nsplit-1]=n$.

m

The total number of eigenvalues (of all $L_i D_i L_i^T$) found.

$needil, neediu$

The indices of the leftmost and rightmost eigenvalues of the root node RRR which are needed to accurately compute the relevant part of the representation tree.

w

Array of size n

The first m elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i D_i L_i^T$, are sorted in ascending order (?larre2a may use the remaining $n-m$ elements as workspace).

Note that immediately after exiting this function, only the eigenvalues in w with indices in range $dol-1:dou-1$ rely on this processor because the eigenvalue computation is done in parallel.

$werr$

Array of size n

The error bound on the corresponding eigenvalue in w .

Note that immediately after exiting this function, only the uncertainties in $werr$ with indices in range $dol-1:dou-1$ are reliable on this processor because the eigenvalue computation is done in parallel.

$wgap$

Array of size n

The separation from the right neighbor eigenvalue in w . The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree.

Exception: at the right end of a block we store the left gap

Note that immediately after exiting this function, only the gaps in $wgap$ with indices in range $dol-1:dou-1$ are reliable on this processor because the eigenvalue computation is done in parallel.

$iblock$

Array of size n

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in w ; $iblock[i]=1$ if eigenvalue $w[i]$ belongs to the first block from the top, $iblock[i]=2$ if $w[i]$ belongs to the second block, and so on.

$indexw$

Array of size n

The indices of the eigenvalues within each block (submatrix); for example, $indexw[i]=10$ and $iblock[i]=2$ imply that the $(i+1)$ -th eigenvalue $w[i]$ is the 10th eigenvalue in block 2.

$gers$

Array of size $2*n$

The n Gerschgorin intervals (the i -th Gerschgorin interval is $(gers[2*i-2], gers[2*i-1])$).

$pivmin$

The minimum pivot in the sturm sequence for T .

$info$

= 0: successful exit

> 0: A problem occurred in ?larre2a.

< 0: One of the called functions signaled an internal problem. Needs inspection of the corresponding parameter *info* for further information.

== -1: Problem in ?larrd2.

== -2: Not enough internal iterations to find base representation.

== -3: Problem in ?larreb2 when computing the refined root representation.

== -4: Problem in ?larreb2 when performing bisection on the desired part of the spectrum.

= -9 Problem: $m < \text{dou-dol} + 1$, that is the code found fewer eigenvalues than it was supposed to.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larrf2

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax

```
void slarrf2(MKL_INT* n, float* d, float* l, float* ld, MKL_INT* clstrt, MKL_INT* clend,
MKL_INT* clmid1, MKL_INT* clmid2, float* w, float* wgap, float* werr, MKL_INT* trymid,
float* spdiam, float* clgapl, float* clgapr, float* pivmin, float* sigma, float* dplus,
float* lplus, float* work, MKL_INT* info);
```

```
void dlarrf2(MKL_INT* n, double* d, double* l, double* ld, MKL_INT* clstrt, MKL_INT*
clend, MKL_INT* clmid1, MKL_INT* clmid2, double* w, double* wgap, double* werr, MKL_INT*
trymid, double* spdiam, double* clgapl, double* clgapr, double* pivmin, double* sigma,
double* dplus, double* lplus, double* work, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

Given the initial representation LDL^T and its cluster of close eigenvalues (in a relative measure), defined by the indices of the first and last eigenvalues in the cluster, ?larrrf2 finds a new relatively robust representation $LDL^T - \sigma I = L_+ D_+ L_+^T$ such that at least one of the eigenvalues of $L_+ D_+ L_+^T$ is relatively isolated.

This is an enhanced version of ?larrrf that also tries shifts in the middle of the cluster, should there be a large gap, in order to break large clusters into at least two pieces.

Input Parameters

<i>n</i>	The order of the matrix (subblock, if the matrix was split).
<i>d</i>	Array of size <i>n</i> The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> .
<i>l</i>	Array of size <i>n</i> -1 The (<i>n</i> -1) subdiagonal elements of the unit bidiagonal matrix <i>L</i> .

<i>ld</i>	Array of size $n-1$ The $(n-1)$ elements $l[i]*d[i]$.
<i>clstrt</i>	The index of the first eigenvalue in the cluster.
<i>clend</i>	The index of the last eigenvalue in the cluster.
<i>clmid1, clmid2</i>	The index of a middle eigenvalue pair with large gap.
<i>w</i>	Array of size $\geq (clend-clstrt+1)$ The eigenvalue approximations of $LD L^T$ in ascending order. $w[clstrt - 1]$ through $w[clend - 1]$ form the cluster of relatively close eigenvalues.
<i>wgap</i>	Array of size $\geq (clend-clstrt+1)$ The separation from the right neighbor eigenvalue in w .
<i>werr</i>	Array of size $\geq (clend-clstrt+1)$ <i>werr</i> contains the semiwidth of the uncertainty interval of the corresponding eigenvalue approximation in w .
<i>spdiam</i>	Estimate of the spectral diameter obtained from the Gerschgorin intervals
<i>clgapl, clgapr</i>	Absolute gap on each end of the cluster. Set by the calling function to protect against shifts too close to eigenvalues outside the cluster.
<i>pivmin</i>	The minimum pivot allowed in the Sturm sequence.
<i>work</i>	Workspace array of size $2*n$

OUTPUT Parameters

<i>wgap</i>	Contains refined values of its input approximations. Very small gaps are unchanged.
<i>sigma</i>	The shift (σ) used to form $L_+ D_+ L_+^T$.
<i>dplus</i>	Array of size n The n diagonal elements of the diagonal matrix D_+ .
<i>lplus</i>	Array of size $n-1$ The first $(n-1)$ elements of <i>lplus</i> contain the subdiagonal elements of the unit bidiagonal matrix L_+ .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?larrv2

Computes the eigenvectors of the tridiagonal matrix $T = L*D*L^T$ given L , D and the eigenvalues of $L*D*L^T$.

Syntax

```
void slarrv2(MKL_INT* n, float* vl, float* vu, float* d, float* l, float* pivmin,
MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou, MKL_INT* needil, MKL_INT*
neediu, float* minrgp, float* rtoll, float* rtol2, float* w, float* werr, float* wgap,
MKL_INT* iblock, MKL_INT* indexw, float* gers, float* sdiam, float* z, MKL_INT* ldz,
MKL_INT* isuppz, float* work, MKL_INT* iwork, MKL_INT* vstart, MKL_INT* finish,
MKL_INT* maxcls, MKL_INT* ndepth, MKL_INT* parity, MKL_INT* zoffset, MKL_INT* info);

void dlarrv2(MKL_INT* n, double* vl, double* vu, double* d, double* l, double* pivmin,
MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou, MKL_INT* needil, MKL_INT*
neediu, double* minrgp, double* rtoll, double* rtol2, double* w, double* werr, double*
wgap, MKL_INT* iblock, MKL_INT* indexw, double* gers, double* sdiam, double* z, MKL_INT*
ldz, MKL_INT* isuppz, double* work, MKL_INT* iwork, MKL_INT* vstart, MKL_INT* finish,
MKL_INT* maxcls, MKL_INT* ndepth, MKL_INT* parity, MKL_INT* zoffset, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

?larrv2 computes the eigenvectors of the tridiagonal matrix $T = LDL^T$ given L , D and approximations to the eigenvalues of LDL^T . The input eigenvalues should have been computed by [larre2a](#) or by previous calls to ?larrv2.

The major difference between the parallel and the sequential construction of the representation tree is that in the parallel case, not all eigenvalues of a given cluster might be computed locally. Other processors might "own" and refine part of an eigenvalue cluster. This is crucial for scalability. Thus there might be communication necessary before the current level of the representation tree can be parsed.

Please note:

- The calling sequence has two additional integer parameters, *dol* and *dou*, that should satisfy $m \geq dou \geq dol \geq 1$. These parameters are only relevant when both eigenvalues and eigenvectors are computed (stegr2b parameter *jobz* = 'V'). ?larrv2 only computes the eigenvectors corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenvectors belonging to $w[0]$ through $w[m-1]$, only the eigenvectors belonging to eigenvalues $w[dol - 1]$ through $w[dou - 1]$ are computed. In this case, only the eigenvalues *dol:dou* are guaranteed to be accurately refined to all figures by Rayleigh-Quotient iteration.
- The additional arguments *vstart*, *finish*, *ndepth*, *parity*, *zoffset* are included as a thread-safe implementation equivalent to save variables. These variables store details about the local representation tree which is computed layerwise. For scalability reasons, eigenvalues belonging to the locally relevant representation tree might be computed on other processors. These need to be communicated before the inspection of the RRRs can proceed on any given layer. Note that only when the variable *finish* is non-zero, the computation has ended. All eigenpairs between *dol* and *dou* have been computed. *m* is set to $dou - dol + 1$.
- ?larrv2 needs more workspace in *z* than the sequential slarrv. It is used to store the conformal embedding of the local representation tree.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

n	The order of the matrix. $n \geq 0$.
vl, vu	Lower and upper bounds of the interval that contains the desired eigenvalues. $vl < vu$. Needed to compute gaps on the left or right end of the extremal eigenvalues in the desired range. vu is currently not used but kept as parameter in case needed.
d	Array of size n The n diagonal elements of the diagonal matrix d . On exit, d is overwritten.
l	Array of size n The $(n-1)$ subdiagonal elements of the unit bidiagonal matrix L are in elements 0 to $n-2$ of l (if the matrix is not split.) At the end of each block is stored the corresponding shift as given by <code>?larre</code> . On exit, l is overwritten.
$pivmin$	The minimum pivot allowed in the sturm sequence.
$isplit$	Array of size n The splitting points, at which the matrix T breaks up into blocks. The first block consists of rows/columns 1 to $isplit[0]$, the second of rows/columns $isplit[0] + 1$ through $isplit[1]$, etc.
m	The total number of input eigenvalues. $0 \leq m \leq n$.
dol, dou	If you want to compute only selected eigenvectors from all the eigenvalues supplied, you can specify an index range $dol:dou$. Or else the setting $dol=1, dou=m$ should be applied. Note that dol and dou refer to the order in which the eigenvalues are stored in w . If you want to compute only selected eigenpairs, the columns $dol-1$ to $dou+1$ of the eigenvector space Z contain the computed eigenvectors. All other columns of Z are set to zero. If $dol > 1$, then $Z(:, dol-1-zoffset)$ is used. If $dou < m$, then $Z(:, dou+1-zoffset)$ is used.
$needil, neediu$	Describe which are the left and right outermost eigenvalues that still need to be included in the computation. These indices indicate whether eigenvalues from other processors are needed to correctly compute the conformally embedded representation tree. When $dol \leq needil \leq neediu \leq dou$, all required eigenvalues are local to the processor and no communication is required to compute its part of the representation tree.
$minrgp$	The minimum relative gap threshold to decide whether an eigenvalue or a cluster boundary is reached.
$rtol1, rtol2$	Parameters for bisection. An interval $[left, right]$ has converged if $right-left < \max(rtol1*gap, rtol2*\max(left , right))$
w	Array of size n

The first m elements of w contain the approximate eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array w from `?stegr2a` is expected here.) Furthermore, they are with respect to the shift of the corresponding root representation for their block.

<i>werr</i>	<p>Array of size n</p> <p>The first m elements contain the semiwidth of the uncertainty interval of the corresponding eigenvalue in w.</p>
<i>wgap</i>	<p>Array of size n</p> <p>The separation from the right neighbor eigenvalue in w.</p>
<i>iblock</i>	<p>Array of size n</p> <p>The indices of the blocks (submatrices) associated with the corresponding eigenvalues in w; $iblock[i]=1$ if eigenvalue $w[i]$ belongs to the first block from the top, $iblock[i]=2$ if $w[i]$ belongs to the second block, and so on.</p>
<i>indexw</i>	<p>Array of size n</p> <p>The indices of the eigenvalues within each block (submatrix). For example: $indexw[i]=10$ and $iblock[i]=2$ imply that the $(i+1)$-th eigenvalue $w[i]$ is the 10th eigenvalue in block 2.</p>
<i>gers</i>	<p>Array of size $2*n$</p> <p>The n Gerschgorin intervals (the i-th Gerschgorin interval is $(gers[2*i-2], gers[2*i-1])$). The Gerschgorin intervals should be computed from the original unshifted matrix.</p> <p>Not used but kept as parameter for possible future use.</p>
<i>sdiam</i>	<p>Array of size n</p> <p>The spectral diameters for all unreduced blocks.</p>
<i>ldz</i>	<p>The leading dimension of the array z. $ldz \geq 1$, and if <code>stegr2b</code> parameter <code>jobz = 'V'</code>, $ldz \geq \max(1, n)$.</p>
<i>work</i>	(workspace) array of size $12*n$
<i>iwork</i>	(workspace) Array of size $7*n$
<i>vstart</i>	Non-zero on initialization, set to zero afterwards.
<i>finish</i>	A flag that indicates whether all eigenpairs have been computed.
<i>maxcls</i>	The largest cluster worked on by this processor in the representation tree.
<i>ndepth</i>	The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.
<i>parity</i>	An internal parameter needed for the storage of the clusters on the current level of the representation tree.
<i>zoffset</i>	Offset for storing the eigenpairs when z is distributed in 1D-cyclic fashion.

OUTPUT Parameters

<i>needil, neediu</i>	
<i>w</i>	Unshifted eigenvalues for which eigenvectors have already been computed.
<i>werr</i>	Contains refined values of its input approximations.
<i>wgap</i>	Contains refined values of its input approximations. Very small gaps are changed.
<i>z</i>	<p>Array of size $ldz * \max(1, m)$</p> <p>If <i>info</i> = 0, the first <i>m</i> columns of the matrix <i>Z</i>, stored in the array <i>z</i>, contain the orthonormal eigenvectors of the matrix <i>T</i> corresponding to the input eigenvalues, with the <i>i</i>-th column of <i>Z</i> holding the eigenvector associated with <i>w</i>[<i>i</i> - 1].</p> <p>In the distributed version, only a subset of columns is accessed, see <i>dol</i>, <i>dou</i>, and <i>zoffset</i>.</p>
<i>isuppz</i>	<p>Array of size $2 * \max(1, m)$</p> <p>The support of the eigenvectors in <i>z</i>, i.e., the indices indicating the non-zero elements in <i>z</i>. The <i>i</i>-th eigenvector is non-zero only in elements <i>isuppz</i>[$2 * i - 2$] through <i>isuppz</i>[$2 * i - 1$].</p>
<i>vstart</i>	Non-zero on initialization, set to zero afterwards.
<i>finish</i>	A flag that indicates whether all eigenpairs have been computed.
<i>maxcls</i>	The largest cluster worked on by this processor in the representation tree.
<i>ndepth</i>	The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.
<i>parity</i>	An internal parameter needed for the storage of the clusters on the current level of the representation tree.
<i>info</i>	<p>= 0: successful exit</p> <p>> 0: A problem occurred in ?larrv2.</p> <p>< 0: One of the called functions signaled an internal problem.</p> <p>Needs inspection of the corresponding parameter <i>info</i> for further information.</p> <p>== -1: Problem in ?larrb2 when refining a child's eigenvalues.</p> <p>== -2: Problem in ?larrf2 when computing the RRR of a child. When a child is inside a tight cluster, it can be difficult to find an RRR. A partial remedy from the user's point of view is to make the parameter <i>minrgp</i> smaller and recompile. However, as the orthogonality of the computed vectors is proportional to $1/minrgp$, be aware that decreasing <i>minrgp</i> might be reduce precision.</p> <p>== -3: Problem in ?larrb2 when refining a single eigenvalue after the Rayleigh correction was rejected.</p> <p>= 5: The Rayleigh Quotient Iteration failed to converge to full accuracy.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?lasorte

Sorts eigenpairs by real and complex data types.

Syntax

```
void slasorte (float *s , MKL_INT *lds , MKL_INT *j , float *out , MKL_INT *info );
void dlasorte (double *s , MKL_INT *lds , MKL_INT *j , double *out , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `?lasorte` function sorts eigenpairs so that real eigenpairs are together and complex eigenpairs are together. This helps to employ 2x2 shifts easily since every second subdiagonal is guaranteed to be zero. This function does no parallel work and makes no calls.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<code>s</code>	(local) Array of size <code>lds</code> . On entry, a matrix already in Schur form.
<code>lds</code>	(local) On entry, the leading dimension of the array <code>s</code> ; unchanged on exit.
<code>j</code>	(local) On entry, the order of the matrix <code>S</code> ; unchanged on exit.
<code>out</code>	(local) Array of size <code>2*j</code> . The work buffer required by the function.
<code>info</code>	(local) Set, if the input matrix had an odd number of real eigenvalues and things could not be paired or if the input matrix <code>S</code> was not originally in Schur form. 0 indicates successful completion.

Output Parameters

<code>s</code>	On exit, the diagonal blocks of <code>S</code> have been rewritten to pair the eigenvalues. The resulting matrix is no longer similar to the input.
<code>out</code>	Work buffer.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?lasrt2

Sorts numbers in increasing or decreasing order.

Syntax

```
void slasrt2 (char *id , MKL_INT *n , float *d , MKL_INT *key , MKL_INT *info );
void dlasrt2 (char *id , MKL_INT *n , double *d , MKL_INT *key , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The ?lasrt2 function is modified LAPACK function [?lasrt](#), which sorts the numbers in d in increasing order (if $id = 'I'$) or in decreasing order (if $id = 'D'$). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20 . The size of `STACK` limits n to about 2^{32} .

Input Parameters

id	<p>= 'I': sort d in increasing order;</p> <p>= 'D': sort d in decreasing order.</p>
n	The length of the array d .
d	<p>Array of size n.</p> <p>On entry, the array to be sorted.</p>
key	<p>Array of size n.</p> <p>On entry, key contains a key to each of the entries in d.</p> <p>Typically, $key[i] = i+1$ for all $i = 0, \dots, n-1$.</p>

Output Parameters

d	<p>On exit, d has been sorted into increasing order $(d[0] \leq \dots \leq d[n-1])$ or into decreasing order $(d[0] \geq \dots \geq d[n-1])$, depending on id.</p>
$info$	<p>= 0: successful exit</p> <p>< 0: if $info = -i$, the i-th argument had an illegal value.</p>
key	<p>On exit, key is permuted in exactly the same manner as d was permuted from input to output. Therefore, if $key[i] = i+1$ for all $i = 0, \dots, n-1$ on input, $d[i]$ on output equals $d[key[i]-1]$ on input.</p>

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?stegr2

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

```
void sstegr2(char* jobz, char* range, MKL_INT* n, float* d, float* e, float* vl, float*
vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m, float* w, float* z, MKL_INT* ldz, MKL_INT*
nzc, MKL_INT* isuppz, float* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork,
MKL_INT* dol, MKL_INT* dou, MKL_INT* zoffset, MKL_INT* info);
```

```
void dstegr2(char* jobz, char* range, MKL_INT* n, double* d, double* e, double* vl,
double* vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m, double* w, double* z, MKL_INT* ldz,
MKL_INT* nzc, MKL_INT* isuppz, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT*
liwork, MKL_INT* dol, MKL_INT* dou, MKL_INT* zoffset, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

?stegr2 computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . It is invoked in the ScaLAPACK MRRR driver p?syevr and the corresponding Hermitian version either when only eigenvalues are to be computed, or when only a single processor is used (the sequential-like case).

?stegr2 has been adapted from LAPACK's ?stegr. Please note the following crucial changes.

1. The calling sequence has two additional integer parameters, *dol* and *dou*, that should satisfy $m \geq dou \geq dol \geq 1$. ?stegr2 only computes the eigenpairs corresponding to eigenvalues *dol* through *dou* in *w*, indexed *dol*-1 through *dou*-1. (That is, instead of computing the eigenpairs belonging to $w[0]$ through $w[m-1]$, only the eigenvectors belonging to eigenvalues $w[dol-1]$ through $w[dou-1]$ are computed. In this case, only the eigenvalues *dol* through *dou* are guaranteed to be fully accurate.
2. *m* is *not* the number of eigenvalues specified by *range*, but is $m = dou - dol + 1$. This concerns the case where only eigenvalues are computed, but on more than one processor. Thus, in this case *m* refers to the number of eigenvalues computed on this processor.
3. The arrays *w* and *z* might not contain all the wanted eigenpairs locally, instead this information is distributed over other processors.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>jobz</i>	= 'N': Compute eigenvalues only; = 'V': Compute eigenvalues and eigenvectors.
<i>range</i>	= 'A': all eigenvalues will be found. = 'V': all eigenvalues in the half-open interval (<i>vl</i> , <i>vu</i>] will be found. = 'I': eigenvalues of the entire matrix with the indices in a given range will be found.

<i>n</i>	The order of the matrix. $n \geq 0$.
<i>d</i>	Array of size <i>n</i> On entry, the <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i> . Overwritten on exit.
<i>e</i>	Array of size <i>n</i> On entry, the (<i>n</i> -1) subdiagonal elements of the tridiagonal matrix <i>T</i> in elements 0 to <i>n</i> -2 of <i>e</i> . <i>e</i> [<i>n</i> -1] need not be set on input, but is used internally as workspace. Overwritten on exit.
<i>vl</i>	If <i>range</i> ='V', the lower and upper bounds of the interval to be searched for eigenvalues. $vl < vu$. Not referenced if <i>range</i> = 'A' or 'I'.
<i>vu</i>	
<i>il, iu</i>	If <i>range</i> ='I', the indices (in ascending order) of the smallest eigenvalue, to be returned in <i>w</i> [<i>il</i> -1], and largest eigenvalue, to be returned in <i>w</i> [<i>iu</i> -1]. $1 \leq il \leq iu \leq n$, if $n > 0$. Not referenced if <i>range</i> = 'A' or 'V'.
<i>ldz</i>	The leading dimension of the array <i>z</i> . $ldz \geq 1$, and if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.
<i>nzc</i>	The number of eigenvectors to be held in the array <i>z</i> , storing the matrix <i>Z</i> . If <i>range</i> = 'A', then $nzc \geq \max(1, n)$. If <i>range</i> = 'V', then $nzc \geq$ the number of eigenvalues in (<i>vl</i> , <i>vu</i>]. If <i>range</i> = 'I', then $nzc \geq iu - il + 1$. If <i>nzc</i> = -1, then a workspace query is assumed; the function calculates the number of columns of the matrix <i>Z</i> that are needed to hold the eigenvectors. This value is returned as the first entry of the <i>z</i> array, and no error message related to <i>nzc</i> is issued.
<i>lwork</i>	The size of the array <i>work</i> . $lwork \geq \max(1, 18 * n)$ if <i>jobz</i> = 'V', and $lwork \geq \max(1, 12 * n)$ if <i>jobz</i> = 'N'. If <i>lwork</i> = -1, then a workspace query is assumed; the function only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued.
<i>liwork</i>	The size of the array <i>iwork</i> . $liwork \geq \max(1, 10 * n)$ if the eigenvectors are desired, and $liwork \geq \max(1, 8 * n)$ if only the eigenvalues are to be computed. If <i>liwork</i> = -1, then a workspace query is assumed; the function only calculates the optimal size of the <i>iwork</i> array, returns this value as the first entry of the <i>iwork</i> array, and no error message related to <i>liwork</i> is issued.
<i>dol, dou</i>	From the eigenvalues <i>w</i> [0] through <i>w</i> [<i>m</i> -1], only eigenvectors <i>Z</i> (:, <i>dol</i>) to <i>Z</i> (:, <i>dou</i>) are computed. If <i>dol</i> > 1, then <i>Z</i> (:, <i>dol</i> -1- <i>zoffset</i>) is used and overwritten.

If $dou < m$, then $Z(:,dou+1-zoffset)$ is used and overwritten.

zoffset

Offset for storing the eigenpairs when z is distributed in 1D-cyclic fashion

OUTPUT Parameters

m

Globally summed over all processors, m equals the total number of eigenvalues found. $0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu - il + 1$. The local output equals $m = dou - dol + 1$.

w

Array of size n

The first m elements contain the selected eigenvalues in ascending order. Note that immediately after exiting this function, only the eigenvalues indexed $dol-1$ through $dou-1$ are reliable on this processor because the eigenvalue computation is done in parallel. Other processors will hold reliable information on other parts of the w array. This information is communicated in the ScaLAPACK driver.

z

Array of size $ldz * \max(1,m)$.

If $jobz = 'V'$, and if $info = 0$, then the first m columns of the matrix Z stored in z contain some of the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i -th column of Z holding the eigenvector associated with $w[i-1]$.

If $jobz = 'N'$, then z is not referenced.

Note: the user must ensure that at least $\max(1,m)$ columns of the matrix are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and can be computed with a workspace query by setting $nzc = -1$, see below.

isuppz

array of size $2 * \max(1,m)$

The support of the eigenvectors in z , i.e., the indices indicating the nonzero elements in z . The i -th computed eigenvector is nonzero only in elements $isuppz[2*i-2]$ through $isuppz[2*i-1]$. This is relevant in the case when the matrix is split. $isuppz$ is only set if $n > 2$.

work

On exit, if $info = 0$, $work[0]$ returns the optimal (and minimal) $lwork$.

iwork

On exit, if $info = 0$, $iwork[0]$ returns the optimal $liwork$.

info

On exit, $info$

= 0: successful exit

other: if $info = -i$, the i -th argument had an illegal value

if $info = 10X$, internal error in ?larre2,

if $info = 20X$, internal error in ?larrv.

Here, the digit $X = \text{ABS}(iinfo) < 10$, where $iinfo$ is the nonzero error code returned by ?larre2 or ?larrv, respectively.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?stegr2a

Computes selected eigenvalues and initial representations needed for eigenvector computations.

Syntax

```
void sstegr2a(char* jobz, char* range, MKL_INT* n, float* d, float* e, float* vl, float*
vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m, float* w, float* z, MKL_INT* ldz, MKL_INT*
nzc, float* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* dol,
MKL_INT* dou, MKL_INT* needil, MKL_INT* neediu, MKL_INT* inderr, MKL_INT* nsplit,
float* pivmin, float* scale, float* wl, float* wu, MKL_INT* info);
```

```
void dstegr2a(char* jobz, char* range, MKL_INT* n, double* d, double* e, double* vl,
double* vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m, double* w, double* z, MKL_INT* ldz,
MKL_INT* nzc, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT*
dol, MKL_INT* dou, MKL_INT* needil, MKL_INT* neediu, MKL_INT* inderr, MKL_INT* nsplit,
double* pivmin, double* scale, double* wl, double* wu, MKL_INT* info);
```

Include Files

- `mk1_scalapack.h`

Description

?stegr2a computes selected eigenvalues and initial representations needed for eigenvector computations in ?stegr2b. It is invoked in the ScaLAPACK MRRR driver p?syevr and the corresponding Hermitian version when both eigenvalues and eigenvectors are computed in parallel on multiple processors. For this case, ?stegr2a implements the first part of the MRRR algorithm, parallel eigenvalue computation and finding the root RRR. At the end of ?stegr2a, other processors might have a part of the spectrum that is needed to continue the computation locally. Once this eigenvalue information has been received by the processor, the computation can then proceed by calling the second part of the parallel MRRR algorithm, ?stegr2b.

Please note:

- The calling sequence has two additional integer parameters, (compared to LAPACK's `stegr`), these are `dol` and `dou` and should satisfy $m \geq dou \geq dol \geq 1$. These parameters are only relevant for the case `jobz = 'V'`.

Globally invoked over all processors, ?stegr2a computes all the eigenvalues specified by `range`.

?stegr2a locally only computes the eigenvalues corresponding to eigenvalues `dol` through `dou` in `w`, indexed `dol-1` through `dou-1`. (That is, instead of computing the eigenvectors belonging to `w[0]` through `w[m-1]`, only the eigenvectors belonging to eigenvalues `w[dol-1]` through `w[dou-1]` are computed. In this case, only the eigenvalues `dol` through `dou` are guaranteed to be fully accurate.

- `m` is not the number of eigenvalues specified by `range`, but it is $m = dou - dol + 1$. Instead, `m` refers to the number of eigenvalues computed on this processor.
- While no eigenvectors are computed in ?stegr2a itself (this is done later in ?stegr2b), the interface

If `jobz = 'V'` then, depending on `range` and `dol`, `dou`, ?stegr2a might need more workspace in `z` than the original ?stegr. In particular, the arrays `w` and `z` might not contain all the wanted eigenpairs locally, instead this information is distributed over other processors.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>jobz</i>	<p>= 'N': Compute eigenvalues only;</p> <p>= 'V': Compute eigenvalues and eigenvectors.</p>
<i>range</i>	<p>= 'A': all eigenvalues will be found.</p> <p>= 'V': all eigenvalues in the half-open interval $(vl, vu]$ will be found.</p> <p>= 'I': eigenvalues of the entire matrix with the indices in a given range will be found.</p>
<i>n</i>	The order of the matrix. $n \geq 0$.
<i>d</i>	<p>Array of size n</p> <p>The n diagonal elements of the tridiagonal matrix T. Overwritten on exit.</p>
<i>e</i>	<p>Array of size n</p> <p>On entry, the $(n-1)$ subdiagonal elements of the tridiagonal matrix T in elements 0 to $n-2$ of e. $e[n-1]$ need not be set on input, but is used internally as workspace. Overwritten on exit.</p>
<i>vl, vu</i>	<p>If $range='V'$, the lower and upper bounds of the interval to be searched for eigenvalues. $vl < vu$.</p> <p>Not referenced if $range = 'A'$ or 'I'.</p>
<i>il, iu</i>	<p>If $range='I'$, the indices (in ascending order) of the smallest eigenvalue, to be returned in $w[il-1]$, and largest eigenvalue, to be returned in $w[iu-1]$. $1 \leq il \leq iu \leq n$, if $n > 0$.</p> <p>Not referenced if $range = 'A'$ or 'V'.</p>
<i>ldz</i>	The leading dimension of the array z . $ldz \geq 1$, and if $jobz = 'V'$, then $ldz \geq \max(1, n)$.
<i>nzc</i>	<p>The number of eigenvectors to be held in the array z.</p> <p>If $range = 'A'$, then $nzc \geq \max(1, n)$.</p> <p>If $range = 'V'$, then $nzc \geq$ the number of eigenvalues in $(vl, vu]$.</p> <p>If $range = 'I'$, then $nzc \geq iu - il + 1$.</p> <p>If $nzc = -1$, then a workspace query is assumed; the function calculates the number of columns of the matrix stored in array z that are needed to hold the eigenvectors. This value is returned as the first entry of the z array, and no error message related to nzc is issued.</p>
<i>lwork</i>	<p>The size of the array $work$. $lwork \geq \max(1, 18*n)$ if $jobz = 'V'$, and $lwork \geq \max(1, 12*n)$ if $jobz = 'N'$.</p> <p>If $lwork = -1$, then a workspace query is assumed; the function only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued.</p>
<i>liwork</i>	The size of the array $iwork$. $liwork \geq \max(1, 10*n)$ if the eigenvectors are desired, and $liwork \geq \max(1, 8*n)$ if only the eigenvalues are to be computed.

If *liwork* = -1, then a workspace query is assumed; the function only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued.

dol, dou

From all the eigenvalues *w*[0] through *w*[*m*-1], only eigenvalues *w*[*dol*-1] through *w*[*dou*-1] are computed.

OUTPUT Parameters

m

Globally summed over all processors, *m* equals the total number of eigenvalues found. $0 \leq m \leq n$.

If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu-il*+1.

The local output equals *m* = *dou* - *dol* + 1.

w

Array of size *n*

The first *m* elements contain approximations to the selected eigenvalues in ascending order. Note that immediately after exiting this function, only the eigenvalues indexed *dol*-1 through *dou*-1 are reliable on this processor because the eigenvalue computation is done in parallel. The other entries are very crude preliminary approximations. Other processors hold reliable information on these other parts of the *w* array.

This information is communicated in the ScaLAPACK driver.

z

Array of size *ldz* * max(1,*m*).

?stegr2a does not compute eigenvectors, this is done in ?stegr2b. The argument *z* as well as all related other arguments only appear to keep the interface consistent and to signal to the user that this function is meant to be used when eigenvectors are computed.

work

On exit, if *info* = 0, *work*[0] returns the optimal (and minimal) *lwork*.

iwork

On exit, if *info* = 0, *iwork*[0] returns the optimal *liwork*.

needil, neediu

The indices of the leftmost and rightmost eigenvalues needed to accurately compute the relevant part of the representation tree. This information can be used to find out which processors have the relevant eigenvalue information needed so that it can be communicated.

inderr

inderr points to the place in the work space where the eigenvalue uncertainties (errors) are stored.

nsplit

The number of blocks into which *T* splits. $1 \leq nsplit \leq n$.

pivmin

The minimum pivot in the sturm sequence for *T*.

scale

The scaling factor for the tridiagonal *T*.

wl, wu

The interval (*wl*, *wu*] contains all the wanted eigenvalues.

It is either given by the user or computed in ?larre2a.

info

On exit, *info* = 0: successful exit

other: if *info* = -*i*, the *i*-th argument had an illegal value

if *info* = 10*x*, internal error in ?larre2a,

Here, the digit $x = \text{abs}(iinfo) < 10$, where *iinfo* is the nonzero error code returned by ?larre2a.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?stegr2b

From eigenvalues and initial representations computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors.

Syntax

```
void sstegr2b(char* jobz, MKL_INT* n, float* d, float* e, MKL_INT* m, float* w, float*
z, MKL_INT* ldz, MKL_INT* nzc, MKL_INT* isuppz, float* work, MKL_INT* lwork, MKL_INT*
iwork, MKL_INT* liwork, MKL_INT* dol, MKL_INT* dou, MKL_INT* needil, MKL_INT* neediu,
MKL_INT* indwlc, float* pivmin, float* scale, float* wl, float* wu, MKL_INT* vstart,
MKL_INT* finish, MKL_INT* maxcls, MKL_INT* ndepth, MKL_INT* parity, MKL_INT* zoffset,
MKL_INT* info);
```

```
void dstegr2b(char* jobz, MKL_INT* n, double* d, double* e, MKL_INT* m, double* w,
double* z, MKL_INT* ldz, MKL_INT* nzc, MKL_INT* isuppz, double* work, MKL_INT* lwork,
MKL_INT* iwork, MKL_INT* liwork, MKL_INT* dol, MKL_INT* dou, MKL_INT* needil, MKL_INT*
neediu, MKL_INT* indwlc, double* pivmin, double* scale, double* wl, double* wu, MKL_INT*
vstart, MKL_INT* finish, MKL_INT* maxcls, MKL_INT* ndepth, MKL_INT* parity, MKL_INT*
zoffset, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

?stegr2b should only be called after a call to ?stegr2a. From eigenvalues and initial representations computed by ?stegr2a, ?stegr2b computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors. It is potentially invoked multiple times on a given processor because the locally relevant representation tree might depend on spectral information that is "owned" by other processors and might need to be communicated.

Please note:

- The calling sequence has two additional integer parameters, *dol* and *dou*, that should satisfy $m \geq dou \geq dol \geq 1$. These parameters are only relevant for the case *jobz* = 'V'. ?stegr2b only computes the eigenvectors corresponding to eigenvalues *dol* through *dou* in *w*, indexed *dol*-1 through *dou*-1. (That is, instead of computing the eigenvectors belonging to $w[0]$ through $w[m-1]$, only the eigenvectors belonging to eigenvalues $w[dol-1]$ through $w[dou-1]$ are computed. In this case, only the eigenvalues *dol* through *dou* are guaranteed to be accurately refined to all figures by Rayleigh-Quotient iteration.
- The additional arguments *vstart*, *finish*, *ndepth*, *parity*, *zoffset* are included as a thread-safe implementation equivalent to save variables. These variables store details about the local representation tree which is computed layerwise. For scalability reasons, eigenvalues belonging to the locally relevant representation tree might be computed on other processors. These need to be communicated before the inspection of the RRRs can proceed on any given layer. Note that only when the variable *finish* is non-zero, the computation has ended. All eigenpairs between *dol* and *dou* have been computed. *m* is set to $dou - dol + 1$.
- ?stegr2b needs more workspace in *z* than the sequential ?stegr. It is used to store the conformal embedding of the local representation tree.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>jobz</i>	<p>= 'N': Compute eigenvalues only;</p> <p>= 'V': Compute eigenvalues and eigenvectors.</p>
<i>n</i>	The order of the matrix. $n \geq 0$.
<i>d</i>	<p>Array of size n</p> <p>The n diagonal elements of the tridiagonal matrix T. Overwritten on exit.</p>
<i>e</i>	<p>Array of size n</p> <p>The $(n-1)$ subdiagonal elements of the tridiagonal matrix T in elements 0 to $n-2$ of e. $e[n-1]$ need not be set on input, but is used internally as workspace. Overwritten on exit.</p>
<i>m</i>	The total number of eigenvalues found in <code>?stegr2a</code> . $0 \leq m \leq n$.
<i>w</i>	<p>Array of size n</p> <p>The first m elements contain approximations to the selected eigenvalues in ascending order. Note that only the eigenvalues from the locally relevant part of the representation tree, that is all the clusters that include eigenvalues from <i>dol</i> through <i>dou</i>, are reliable on this processor. (It does not need to know about any others anyway.)</p>
<i>ldz</i>	The leading dimension of the array z . $ldz \geq 1$, and if <i>jobz</i> = 'V', then $ldz \geq \max(1, n)$.
<i>nzc</i>	The number of eigenvectors to be held in the array z , storing the matrix Z .
<i>lwork</i>	<p>The size of the array <i>work</i>. $lwork \geq \max(1, 18*n)$</p> <p>if <i>jobz</i> = 'V', and $lwork \geq \max(1, 12*n)$ if <i>jobz</i> = 'N'.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the function only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued.</p>
<i>liwork</i>	<p>The size of the array <i>iwork</i>. $liwork \geq \max(1, 10*n)$ if the eigenvectors are desired, and $liwork \geq \max(1, 8*n)$ if only the eigenvalues are to be computed.</p> <p>If <i>liwork</i> = -1, then a workspace query is assumed; the function only calculates the optimal size of the <i>iwork</i> array, returns this value as the first entry of the <i>iwork</i> array, and no error message related to <i>liwork</i> is issued.</p>
<i>dol, dou</i>	<p>From the eigenvalues $w[0]$ through $w[m-1]$, only eigenvectors $Z(:, dol)$ to $Z(:, dou)$ are computed.</p> <p>If <i>dol</i> > 1, then $Z(:, dol-1-zoffset)$ is used and overwritten.</p>

	If $dou < m$, then $Z(:,dou+1-zoffset)$ is used and overwritten.
<i>needil, neediu</i>	Describes which are the left and right outermost eigenvalues still to be computed. Initially computed by <code>?larre2a</code> , modified in the course of the algorithm.
<i>pivmin</i>	The minimum pivot in the sturm sequence for T .
<i>scale</i>	The scaling factor for T . Used for unscaling the eigenvalues at the very end of the algorithm.
<i>wl, wu</i>	The interval $(wl, wu]$ contains all the wanted eigenvalues.
<i>vstart</i>	Non-zero on initialization, set to zero afterwards.
<i>finish</i>	Indicates whether all eigenpairs have been computed.
<i>maxcls</i>	The largest cluster worked on by this processor in the representation tree.
<i>ndepth</i>	The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.
<i>parity</i>	An internal parameter needed for the storage of the clusters on the current level of the representation tree.
<i>zoffset</i>	Offset for storing the eigenpairs when z is distributed in 1D-cyclic fashion.

OUTPUT Parameters

<i>z</i>	<p>Array of size $ldz * \max(1,m)$</p> <p>If $jobz = 'V'$, and if $info = 0$, then a subset of the first m columns of the matrix Z, stored in z, contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i-th column of Z holding the eigenvector associated with $w[i-1]$.</p> <p>See <i>dol, dou</i> for more information.</p>
<i>isuppz</i>	<p>array of size $2 * \max(1,m)$.</p> <p>The support of the eigenvectors in z, i.e., the indices indicating the nonzero elements in z. The i-th computed eigenvector is nonzero only in elements $isuppz[2*i-2]$ through $isuppz[2*i-1]$. This is relevant in the case when the matrix is split. <i>isuppz</i> is only set if $n > 2$.</p>
<i>work</i>	On exit, if $info = 0$, $work[0]$ returns the optimal (and minimal) $lwork$.
<i>iwork</i>	On exit, if $info = 0$, $iwork[0]$ returns the optimal $liwork$.
<i>needil, neediu</i>	Modified in the course of the algorithm.
<i>indwlc</i>	Pointer into the workspace location where the local eigenvalue representations are stored. ("Local eigenvalues" are those relative to the individual shifts of the RRRs.)
<i>vstart</i>	Non-zero on initialization, set to zero afterwards.
<i>finish</i>	Indicates whether all eigenpairs have been computed
<i>maxcls</i>	The largest cluster worked on by this processor in the representation tree.

<i>ndepth</i>	The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.
<i>parity</i>	An internal parameter needed for the storage of the clusters on the current level of the representation tree.
<i>info</i>	On exit, <i>info</i> = 0: successful exit other: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value if <i>info</i> = 20 <i>x</i> , internal error in ?larrv2 . Here, the digit <i>x</i> = abs(<i>iinfo</i>) < 10, where <i>iinfo</i> is the nonzero error code returned by ?larrv2

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?stein2

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.

Syntax

```
void sstein2 (MKL_INT *n , float *d , float *e , MKL_INT *m , float *w , MKL_INT
*iblock , MKL_INT *isplit , float *orfac , float *z , MKL_INT *ldz , float *work ,
MKL_INT *iwork , MKL_INT *ifail , MKL_INT *info );

void dstein2 (MKL_INT *n , double *d , double *e , MKL_INT *m , double *w , MKL_INT
*iblock , MKL_INT *isplit , double *orfac , double *z , MKL_INT *ldz , double *work ,
MKL_INT *iwork , MKL_INT *ifail , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The [?stein2](#) function is a modified LAPACK function [?stein](#). It computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter *maxits* (currently set to 5).

Input Parameters

<i>n</i>	The order of the matrix T ($n \geq 0$).
<i>m</i>	The number of eigenvectors to be found ($0 \leq m \leq n$).
<i>d</i> , <i>e</i> , <i>w</i>	Arrays: <i>d</i> , of size <i>n</i> . The <i>n</i> diagonal elements of the tridiagonal matrix T . <i>e</i> , of size <i>n</i> . The (<i>n</i> -1) subdiagonal elements of the tridiagonal matrix T , in elements 1 to <i>n</i> -1. <i>e</i> [<i>n</i> -1] need not be set.

w , of size n .

The first m elements of w contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array w from `?stebz` with `ORDER = 'B'` is expected here).

The size of w must be at least $\max(1, n)$.

iblock

Array of size n .

The submatrix indices associated with the corresponding eigenvalues in w ;

$iblock[i] = 1$, if eigenvalue $w[i]$ belongs to the first submatrix from the top,

$iblock[i] = 2$, if eigenvalue $w[i]$ belongs to the second submatrix, etc. (The output array *iblock* from `?stebz` is expected here).

isplit

Array of size n .

The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to $isplit[0]$, the second submatrix consists of rows/columns $isplit[0]+1$ through $isplit[1]$, etc. (The output array *isplit* from `?stebz` is expected here).

orfac

orfac specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues which are within $orfac * ||T||$ of each other are to be orthogonalized.

ldz

The leading dimension of the output array z ; $ldz \geq \max(1, n)$.

work

Workspace array of size $5n$.

iwork

Workspace array of size n .

Output Parameters

z

Array of size $ldz * m$.

The computed eigenvectors. The eigenvector associated with the eigenvalue $w[i]$ is stored in the $(i+1)$ -th column of the matrix Z represented by z , $i=0, \dots, m-1$. Any vector that fails to converge is set to its current iterate after *maxits* iterations.

ifail

Array of size m .

On normal exit, all elements of *ifail* are zero. If one or more eigenvectors fail to converge after *maxits* iterations, then their indices are stored in the array *ifail*.

info

$info = 0$, the exit is successful.

$info < 0$: if $info = -i$, the i -th had an illegal value.

$info > 0$: if $info = i$, then i eigenvectors failed to converge in *maxits* iterations. Their indices are stored in the array *ifail*.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?dbtf2

Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).

Syntax

```
void sdbtf2 (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , float *ab , MKL_INT
*ldab , MKL_INT *info );

void ddbtf2 (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , double *ab , MKL_INT
*ldab , MKL_INT *info );

void cdbtf2 (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , MKL_Complex8 *ab ,
MKL_INT *ldab , MKL_INT *info );

void zdbtf2 (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , MKL_Complex16 *ab ,
MKL_INT *ldab , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `?dbtf2` function computes an LU factorization of a general real/complex m -by- n band matrix A without using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling [BLAS Routines and Functions](#).

Input Parameters

m	The number of rows of the matrix A ($m \geq 0$).
n	The number of columns in A ($n \geq 0$).
kl	The number of sub-diagonals within the band of A ($kl \geq 0$).
ku	The number of super-diagonals within the band of A ($ku \geq 0$).
ab	Array of size $ldab * n$. The matrix A in band storage, in rows $kl+1$ to $2kl+ku+1$; rows 1 to kl of the matrix need not be set. The j -th column of A is stored in the array ab as follows: $ab[kl+ku+i-j+(j-1)*ldab] = A(i,j)$ for $\max(1, j-ku) \leq i \leq \min(m, j+kl)$.
$ldab$	The leading dimension of the array ab . ($ldab \geq 2kl + ku + 1$)

Output Parameters

ab	On exit, details of the factorization: U is stored as an upper triangular band matrix with $kl+ku$ superdiagonals in rows 1 to $kl+ku+1$, and the multipliers used during the factorization are stored in rows $kl+ku+2$ to $2*kl+ku+1$. See the <i>Application Notes</i> below for further details.
$info$	$= 0$: successful exit

< 0: if *info* = - *i*, the *i*-th argument had an illegal value,
 > 0: if *info* = + *i*, the matrix element $U(i,i)$ is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:

on entry

$$\begin{bmatrix} * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{bmatrix}$$

on exit

$$\begin{bmatrix} * & u_{12} & u_{23} & u_{34} & u_{45} \\ u_{11} & u_{22} & u_{33} & u_{44} & u_{55} \\ m_{21} & m_{32} & m_{43} & m_{54} & m_{65} \\ m_{31} & m_{42} & m_{53} & m_{64} & * \end{bmatrix}$$

The function does not use array elements marked *; elements marked + need not be set on entry, but the function requires them to store elements of *U*, because of fill-in resulting from the row interchanges.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?dbtrf

Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).

Syntax

```
void sdbtrf (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , float *ab , MKL_INT
*ldab , MKL_INT *info );

void ddbtrf (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , double *ab , MKL_INT
*ldab , MKL_INT *info );

void cdbtrf (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , MKL_Complex8 *ab ,
MKL_INT *ldab , MKL_INT *info );

void zdbtrf (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , MKL_Complex16 *ab ,
MKL_INT *ldab , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

This function computes an LU factorization of a real m -by- n band matrix *A* without using partial pivoting or row interchanges.

This is the blocked version of the algorithm, calling [BLAS Routines and Functions](#).

Input Parameters

m The number of rows of the matrix *A* ($m \geq 0$).

n	The number of columns in A ($n \geq 0$).
kl	The number of sub-diagonals within the band of A ($kl \geq 0$).
ku	The number of super-diagonals within the band of A ($ku \geq 0$).
ab	Array of size $ldab * n$. The matrix A in band storage, in rows $kl+1$ to $2kl+ku+1$; rows 1 to kl need not be set. The j -th column of A is stored in the array ab as follows: $ab[kl + ku + i - j + (j-1) * ldab] = A(i, j)$ for $\max(1, j - ku) \leq i \leq \min(m, j + kl)$.
$ldab$	The leading dimension of the array ab . ($ldab \geq 2kl + ku + 1$)

Output Parameters

ab	On exit, details of the factorization: U is stored as an upper triangular band matrix with $kl+ku$ superdiagonals in rows 1 to $kl+ku+1$, and the multipliers used during the factorization are stored in rows $kl+ku+2$ to $2*kl+ku+1$. See the <i>Application Notes</i> below for further details.
$info$	= 0: successful exit < 0: if $info = -i$, the i -th argument had an illegal value, > 0: if $info = +i$, the matrix element $U(i,i)$ is 0. The factorization has been completed, but the factor U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:

on entry

$$\begin{bmatrix} * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{bmatrix}$$

on exit

$$\begin{bmatrix} * & u_{12} & u_{23} & u_{34} & u_{45} \\ u_{11} & u_{22} & u_{33} & u_{44} & u_{55} \\ m_{21} & m_{32} & m_{43} & m_{54} & m_{65} \\ m_{31} & m_{42} & m_{53} & m_{64} & * \end{bmatrix}$$

The function does not use array elements marked *.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?dtttrf

Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).

Syntax

```
void sdttrf (MKL_INT *n , float *dl , float *d , float *du , MKL_INT *info );
```

```
void ddttrf (MKL_INT *n , double *dl , double *d , double *du , MKL_INT *info );
void cdttrf (MKL_INT *n , MKL_Complex8 *dl , MKL_Complex8 *d , MKL_Complex8 *du ,
MKL_INT *info );
void zdttrf (MKL_INT *n , MKL_Complex16 *dl , MKL_Complex16 *d , MKL_Complex16 *du ,
MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `?dttrf` function computes an LU factorization of a real or complex tridiagonal matrix A using elimination without partial pivoting.

The factorization has the form $A = L*U$, where L is a product of unit lower bidiagonal matrices and U is upper triangular with nonzeros only in the main diagonal and first superdiagonal.

Input Parameters

n	The order of the matrix A ($n \geq 0$).
dl, d, du	Arrays containing elements of A . The array dl of size $(n-1)$ contains the sub-diagonal elements of A . The array d of size n contains the diagonal elements of A . The array du of size $(n-1)$ contains the super-diagonal elements of A .

Output Parameters

dl	Overwritten by the $(n-1)$ multipliers that define the matrix L from the LU factorization of A .
d	Overwritten by the n diagonal elements of the upper triangular matrix U from the LU factorization of A .
du	Overwritten by the $(n-1)$ elements of the first super-diagonal of U .
$info$	= 0: successful exit < 0: if $info = -i$, the i -th argument had an illegal value, > 0: if $info = i$, the matrix element $U(i,i)$ is exactly 0. The factorization has been completed, but the factor U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?dttrsv

Solves a general tridiagonal system of linear equations using the LU factorization computed by `?dttrf`.

Syntax

```
void sdttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , float *dl , float
*d , float *du , float *b , MKL_INT *ldb , MKL_INT *info );
```



```

void ddttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , double *dl ,
double *d , double *du , double *b , MKL_INT *ldb , MKL_INT *info );

void cdttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8
*d , MKL_Complex8 *d , MKL_Complex8 *du , MKL_Complex8 *b , MKL_INT *ldb , MKL_INT
*info );

void zdttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16
*d , MKL_Complex16 *d , MKL_Complex16 *du , MKL_Complex16 *b , MKL_INT *ldb , MKL_INT
*info );

```

Include Files

- mkl_scalapack.h

Description

The `ddttrsv` function solves one of the following systems of linear equations:

$$L * X = B, L^T * X = B, \text{ or } L^H * X = B,$$

$$U * X = B, U^T * X = B, \text{ or } U^H * X = B$$

with factors of the tridiagonal matrix A from the LU factorization computed by `ddttrf`.

Input Parameters

<i>uplo</i>	Specifies whether to solve with L or U .
<i>trans</i>	Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $A * X = B$ is solved for X (no transpose). If <i>trans</i> = 'T', then $A^T * X = B$ is solved for X (transpose). If <i>trans</i> = 'C', then $A^H * X = B$ is solved for X (conjugate transpose).
<i>n</i>	The order of the matrix A ($n \geq 0$).
<i>nrhs</i>	The number of right-hand sides, that is, the number of columns in the matrix B ($nrhs \geq 0$).
<i>dl,d,du,b</i>	The array <i>dl</i> of size $(n - 1)$ contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A . The array <i>d</i> of size n contains n diagonal elements of the upper triangular matrix U from the LU factorization of A . The array <i>du</i> of size $(n - 1)$ contains the $(n - 1)$ elements of the first super-diagonal of U . On entry, the array <i>b</i> of size $ldb * nrhs$ contains the right-hand side of matrix B .
<i>ldb</i>	The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	If <i>info</i> =0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?pttrsv

*Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the $L^*D^*L^H$ factorization computed by ?pttrf.*

Syntax

```
void spttrsv (char *trans , MKL_INT *n , MKL_INT *nrhs , float *d , float *e , float
*b , MKL_INT *ldb , MKL_INT *info );

void dpttrsv (char *trans , MKL_INT *n , MKL_INT *nrhs , double *d , double *e , double
*b , MKL_INT *ldb , MKL_INT *info );

void cpttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , float *d ,
MKL_Complex8 *e , MKL_Complex8 *b , MKL_INT *ldb , MKL_INT *info );

void zpttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , double *d ,
MKL_Complex16 *e , MKL_Complex16 *b , MKL_INT *ldb , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The ?pttrsv function solves one of the triangular systems:

$L^T * X = B$, or $L * X = B$ for real flavors,

or

$L * X = B$, or $L^H * X = B$,

$U * X = B$, or $U^H * X = B$ for complex flavors,

where L (or U for complex flavors) is the Cholesky factor of a Hermitian positive-definite tridiagonal matrix A such that

$A = L^*D^*L^H$ (computed by [spttrf/dpttrf](#))

or

$A = U^H * D * U$ or $A = L^*D^*L^H$ (computed by [cpttrf/zpttrf](#)).

Input Parameters

uplo

Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and the form of the factorization:

If $uplo = 'U'$, e is the superdiagonal of U , and $A = U^H * D * U$ or $A = L^*D^*L^H$;

if $uplo = 'L'$, e is the subdiagonal of L , and $A = L^*D^*L^H$.

The two forms are equivalent, if A is real.

<i>trans</i>	Specifies the form of the system of equations: for real flavors: if <i>trans</i> = 'N': $L^T X = B$ (no transpose) if <i>trans</i> = 'T': $L^T X = B$ (transpose) for complex flavors: if <i>trans</i> = 'N': $U^H X = B$ or $L^H X = B$ (no transpose) if <i>trans</i> = 'C': $U^H X = B$ or $L^H X = B$ (conjugate transpose).
<i>n</i>	The order of the tridiagonal matrix <i>A</i> . $n \geq 0$.
<i>nrhs</i>	The number of right hand sides, that is, the number of columns of the matrix <i>B</i> . $nrhs \geq 0$.
<i>d</i>	array of size <i>n</i> . The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> from the factorization computed by ?pttrf .
<i>e</i>	array of size $(n-1)$. The $(n-1)$ off-diagonal elements of the unit bidiagonal factor <i>U</i> or <i>L</i> from the factorization computed by ?pttrf . See <i>uplo</i> .
<i>b</i>	array of size <i>ldb</i> * <i>nrhs</i> . On entry, the right hand side matrix <i>B</i> .
<i>ldb</i>	The leading dimension of the array <i>b</i> . $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	On exit, the solution matrix <i>X</i> .
<i>info</i>	= 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?steqr2

Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

Syntax

```
void ssteqr2 (char *compz , MKL_INT *n , float *d , float *e , float *z , MKL_INT *ldz ,
MKL_INT *nr , float *work , MKL_INT *info );

void dsteqr2 (char *compz , MKL_INT *n , double *d , double *e , double *z , MKL_INT
*ldz , MKL_INT *nr , double *work , MKL_INT *info );
```

Include Files

- `mkl_scalapack.h`

Description

The `?steqr2` function is a modified version of LAPACK function `?steqr`. The `?steqr2` function computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. `?steqr2` is modified from `?steqr` to allow each ScaLAPACK process running `?steqr2` to perform updates on a distributed matrix Q . Proper usage of `?steqr2` can be gleaned from examination of ScaLAPACK function `p?syev`.

Input Parameters

<i>compz</i>	<p>Must be 'N' or 'I'.</p> <p>If <i>compz</i> = 'N', the function computes eigenvalues only. If <i>compz</i> = 'I', the function computes the eigenvalues and eigenvectors of the tridiagonal matrix T.</p> <p>z must be initialized to the identity matrix by <code>p?laset</code> or <code>?laset</code> prior to entering this function.</p>
<i>n</i>	The order of the matrix T ($n \geq 0$).
<i>d, e, work</i>	<p>Arrays:</p> <p>d contains the diagonal elements of T. The size of d must be at least $\max(1, n)$.</p> <p>e contains the $(n-1)$ subdiagonal elements of T. The size of e must be at least $\max(1, n-1)$.</p> <p>$work$ is a workspace array. The size of $work$ is $\max(1, 2*n-2)$. If <i>compz</i> = 'N', then $work$ is not referenced.</p>
<i>z</i>	<p>(local)</p> <p>Array of global size $n * n$ and of local size $ldz * nr$.</p> <p>If <i>compz</i> = 'V', then z contains the orthogonal matrix used in the reduction to tridiagonal form.</p>
<i>ldz</i>	<p>The leading dimension of the array z. Constraints:</p> <p>$ldz \geq 1$,</p> <p>$ldz \geq \max(1, n)$, if eigenvectors are desired.</p>
<i>nr</i>	<p>$nr = \max(1, \text{numroc}(n, nb, myprow, 0, nprocs))$.</p> <p>If <i>compz</i> = 'N', then nr is not referenced.</p>

Output Parameters

<i>d</i>	<p>On exit, the eigenvalues in ascending order, if <i>info</i> = 0.</p> <p>See also <i>info</i>.</p>
<i>e</i>	On exit, e has been destroyed.
<i>z</i>	On exit, if <i>info</i> = 0, then,

if `compz = 'V'`, `z` contains the orthonormal eigenvectors of the original symmetric matrix, and if `compz = 'I'`, `z` contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If `compz = 'N'`, then `z` is not referenced.

`info`

`info = 0`, the exit is successful.

`info < 0`: if `info = -i`, the i -th had an illegal value.

`info > 0`: the algorithm has failed to find all the eigenvalues in a total of $30n$ iterations;

if `info = i`, then i elements of e have not converged to zero; on exit, d and e contain the elements of a symmetric tridiagonal matrix, which is orthogonally similar to the original matrix.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

?trmvt

Performs matrix-vector operations.

Syntax

```
void strmvt (const char* uplo, const MKL_INT* n, const float* t, const MKL_INT* ldt,
float* x, const MKL_INT* incx, const float* y, const MKL_INT* incy, float* w, const
MKL_INT* incw, const float* z, const MKL_INT* incz);

void dtrmvt (const char* uplo, const MKL_INT* n, const double* t, const MKL_INT* ldt,
double* x, const MKL_INT* incx, const double* y, const MKL_INT* incy, double* w, const
MKL_INT* incw, const double* z, const MKL_INT* incz);

void ctrmvt (const char* uplo, const MKL_INT* n, const MKL_Complex8* t, const MKL_INT*
ldt, MKL_Complex8* x, const MKL_INT* incx, const MKL_Complex8* y, const MKL_INT* incy,
MKL_Complex8* w, const MKL_INT* incw, const MKL_Complex8* z, const MKL_INT* incz);

void ztrmvt (const char* uplo, const MKL_INT* n, const MKL_Complex16* t, const MKL_INT*
ldt, MKL_Complex16* x, const MKL_INT* incx, const MKL_Complex16* y, const MKL_INT*
incy, MKL_Complex16* w, const MKL_INT* incw, const MKL_Complex16* z, const MKL_INT*
incz);
```

Include Files

- `mkl_scalapack.h`

Description

?trmvt performs the matrix-vector operations as follows:

strmvt and dtrmvt: $x := T^T * y$, and $w := T * z$

ctrmvt and ztrmvt: $x := \text{conjg}(T^T) * y$, and $w := T * z$,

where x is an n element vector and T is an n -by- n upper or lower triangular matrix.

Input Parameters

`uplo`

On entry, `uplo` specifies whether the matrix is an upper or lower triangular matrix as follows:

uplo = 'U' or 'u'

A is an upper triangular matrix.

uplo = 'L' or 'l'

A is a lower triangular matrix.

Unchanged on exit.

n

On entry, *n* specifies the order of the matrix A. *n* must be at least zero.

Unchanged on exit.

t

Array of size (*ldt*, *n*).

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

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

lda

On entry, *lda* specifies the first dimension of A as declared in the calling (sub) program. *lda* must be at least max(1, *n*).

Unchanged on exit.

incx

On entry, *incx* specifies the increment for the elements of *x*. *incx* must not be zero.

Unchanged on exit.

y

Array of size at least (1 + (*n* - 1) * abs(*incy*)).

Before entry, the incremented array *y* must contain the *n* element vector *y*.

Unchanged on exit.

incy

On entry, *incy* specifies the increment for the elements of *y*. *incy* must not be zero.

Unchanged on exit.

incw

On entry, *incw* specifies the increment for the elements of *w*. *incw* must not be zero.

Unchanged on exit.

z

Array of size at least (1 + (*n* - 1) * abs(*incz*)).

Before entry, the incremented array *z* must contain the *n* element vector *z*.

Unchanged on exit.

incz

On entry, *incz* specifies the increment for the elements of *z*. *incz* must not be zero.

Unchanged on exit.

Output Parameters

<i>t</i>	<p>Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i>-by-<i>n</i> upper triangular part of the array <i>t</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>t</i> is not referenced.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i>-by-<i>n</i> lower triangular part of the array <i>t</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>t</i> is not referenced.</p>
<i>x</i>	<p>Array of size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$.</p> <p>On exit, $x = T * y$.</p>
<i>w</i>	<p>Array of size at least $(1 + (n - 1) * \text{abs}(\text{incw}))$.</p> <p>On exit, $w = T * z$.</p>

pilaenv

Returns the positive integer value of the logical blocking size.

Syntax

```
MKL_INT pilaenv (const MKL_INT *ictxt , const char *prec);
```

Include Files

- mkl_pblas.h

Description

pilaenv returns the positive integer value of the logical blocking size. This value is machine and precision specific. This version provides a logical blocking size which should give good though not optimal performance on many of the currently available distributed-memory concurrent computers. You are encouraged to modify this subroutine to set this tuning parameter for your particular machine.

Input Parameters

<i>ictxt</i>	On entry, <i>ictxt</i> specifies the BLACS context handle, indicating the global context of the operation. The context itself is global, but the value of <i>ictxt</i> is local.
<i>prec</i>	<p>On input, <i>prec</i> specifies the precision for which the logical block size should be returned as follows:</p> <p><i>prec</i> = 'S' or 's' single precision real,</p> <p><i>prec</i> = 'D' or 'd' double precision real,</p> <p><i>prec</i> = 'C' or 'c' single precision complex,</p> <p><i>prec</i> = 'Z' or 'z' double precision complex,</p> <p><i>prec</i> = 'I' or 'i' integer.</p>

Application Notes

Before modifying this routine to tune the library performance on your system, be aware of the following:

1. The value this function returns must be strictly larger than zero,

2. If you are planning to link your program with different instances of the library (for example, on a heterogeneous machine), you *must* compile each instance of the library with exactly the same version of this routine for obvious interoperability reasons.

pilaenvx

Called from the ScaLAPACK routines to choose problem-dependent parameters for the local environment.

Syntax

```
MKL_INT pilaenvx (const MKL_INT* ictxt, const MKL_INT* ispec, const char* name, const
char* opts, const MKL_INT* n1, const MKL_INT* n2, const MKL_INT* n3, const MKL_INT*
```

Include Files

- mkl.h

Description

`pilaenvx` is called from the ScaLAPACK routines to choose problem-dependent parameters for the local environment. See `ispec` for a description of the parameters. This version provides a set of parameters which should give good, though not optimal, performance on many of the currently available computers. You are encouraged to modify this subroutine to set the tuning parameters for your particular machine using the option and problem size information in the arguments.

Input Parameters

<code>ictxt</code>	(local input) On entry, <code>ictxt</code> specifies the BLACS context handle, indicating the global context of the operation. The context itself is global, but the value of <code>ictxt</code> is local.
<code>ispec</code>	(global input) Specifies the parameter to be returned as the value of <code>pilaenvx</code> . = 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance (unlikely). = 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used. = 3: the crossover point (in a block routine, for N less than this value, an unblocked routine should be used). = 4: the number of shifts, used in the nonsymmetric eigenvalue routines (DEPRECATED). = 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least k by m , where k is given by <code>pilaenvx(2,...)</code> and m by <code>pilaenvx(5,...)</code> . = 6: the crossover point for the SVD (when reducing an m by n matrix to bidiagonal form, if $\max(m,n)/\min(m,n)$ exceeds this value, a QR factorization is used first to reduce the matrix to a triangular form). = 7: the number of processors. = 8: the crossover point for the multishift QR method for nonsymmetric eigenvalue problems (DEPRECATED).

= 9: maximum size of the subproblems at the bottom of the computation tree in the divide-and-conquer algorithm (used by `?gelsd` and `?gesdd`).

=10: IEEE NaN arithmetic can be trusted not to trap.

=11: infinity arithmetic can be trusted not to trap.

12 <= *ispec* <= 16:

`p?hseqr` or one of its subroutines, see `piparmq` for detailed explanation.

17 <= *ispec* <= 22:

Parameters for `pb?trord/p?hseqr` (not all), as follows:

=17: maximum number of concurrent computational windows;

=18: number of eigenvalues/bulges in each window;

=19: computational window size;

=20: minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations;

=21: width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form;

=22: the maximum number of eigenvalues moved together over a process border;

=23: the number of processors involved in Aggressive Early Deflation (AED);

=99: Maximum iteration chunksize in OpenMP parallelization.

name

(global input)

The name of the calling subroutine, in either upper case or lower case.

opts

(global input) The character options to the subroutine name, concatenated into a single character string. For example, *uplo* = 'U', *trans* = 'T', and *diag* = 'N' for a triangular routine would be specified as *opts* = 'UTN'.

n1, *n2*, *n3*, and *n4*

(global input) Problem dimensions for the subroutine name; these may not all be required.

Output Parameters

result

(global output)

>= 0: the value of the parameter specified by *ispec*.

< 0: if *pilaenvx* = -*k*, the *k*-th argument had an illegal value.

Application Notes

The following conventions have been used when calling `ilaenv` from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine name, in the same order that they appear in the argument list for name, even if they are not used in determining the value of the parameter specified by *ispec*.

2. The problem dimensions $n1$, $n2$, $n3$, and $n4$ are specified in the order that they appear in the argument list for *name*. $n1$ is used first, $n2$ second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by `ilaenv` is checked for validity in the calling subroutine. For example, `ilaenv` is used to retrieve the optimal block size for `strtri` as follows:

```
NB = ilaenv( 1, 'STRTRI', UPLO // DIAG, N, -1, -1, -1 );
if( NB<=1 ) {
    NB = MAX( 1, N );
}
```

The same conventions hold for this ScaLAPACK-style variant.

pjlaenv

Called from the ScaLAPACK symmetric and Hermitian tailored eigen-routines to choose problem-dependent parameters for the local environment.

Syntax

```
MKL_INT pjlaenv (const MKL_INT* ictxt, const MKL_INT* ispec, const char* name, const
char* opts, const MKL_INT* n1, const MKL_INT* n2, const MKL_INT* n3, const MKL_INT*
n4);
```

Include Files

- `mkl.h`

Description

`pjlaenv` is called from the ScaLAPACK symmetric and Hermitian tailored eigen-routines to choose problem-dependent parameters for the local environment. See *ispec* for a description of the parameters. This version provides a set of parameters which should give good, though not optimal, performance on many of the currently available computers. You are encouraged to modify this subroutine to set the tuning parameters for your particular machine using the option and problem size information in the arguments.

Input Parameters

<i>ispec</i>	(global input) Specifies the parameter to be returned as the value of <code>pjlaenv</code> . = 1: the data layout blocksize; = 2: the panel blocking factor; = 3: the algorithmic blocking factor; = 4: execution path control; = 5: maximum size for direct call to the LAPACK routine.
<i>name</i>	(global input) The name of the calling subroutine, in either upper case or lower case.
<i>opts</i>	(global input) The character options to the subroutine name, concatenated into a single character string. For example, <i>uplo</i> = 'U', <i>trans</i> = 'T', and <i>diag</i> = 'N' for a triangular routine would be specified as <i>opts</i> = 'UTN'.

n1, n2, n3, and n4

(global input) Problem dimensions for the subroutine name; these may not all be required. At present, only *n1* is used, and it (*n1*) is used only for 'TTRD'.

Output Parameters

result

(global or local output)

≥ 0 : the value of the parameter specified by *ispec*.

< 0 : if *pjlaenv* = $-k$, the k -th argument had an illegal value. Most parameters set via a call to *pjlaenv* must be identical on all processors and hence *pjlaenv* will return the same value to all procesors (i.e. global output). However some, in particular, the panel blocking factor can be different on each processor and hence *pjlaenv* can return different values on different processors (i.e. local output).

Application Notes

The following conventions have been used when calling *pjlaenv* from the ScaLAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine name, in the same order that they appear in the argument list for name, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1, n2, n3, and n4* are specified in the order that they appear in the argument list for name. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
 - a. The parameter value returned by *pjlaenv* is checked for validity in the calling subroutine. For example, *pjlaenv* is used to retrieve the optimal blocksize for STRTRI as follows:

```
NB = pjlaenv( 1, 'STRTRI', UPLO // DIAG, N, -1, -1, -1 );
IF( NB>=1 ) {
    NB = MAX( 1, N );
}
```

pjlaenv is patterned after *ilaenv* and keeps the same interface in anticipation of future needs, even though *pjlaenv* is only sparsely used at present in ScaLAPACK. Most ScaLAPACK codes use the input data layout blocking factor as the algorithmic blocking factor - hence there is no need or opportunity to set the algorithmic or data decomposition blocking factor. *pXYTevx.f* and *pXYTgvx.f* and *pXYTtrd.f* are the only codes which call *pjlaenv*. *pXYTevx.f* and *pXYTgvx.f* redistribute the data to the best data layout for each transformation. *pXYTtrd.f* uses a data layout blocking factor of 1.

Additional ScaLAPACK Routines

```
void pchetttrd (const char *uplo , const MKL_INT *n , MKL_Complex8 *a , const MKL_INT
*ia , const MKL_INT *ja , const MKL_INT *desca , float *d , float *e , MKL_Complex8
*tau , MKL_Complex8 *work , const MKL_INT *lwork , MKL_INT *info );

void pzhettrd (const char *uplo , const MKL_INT *n , MKL_Complex16 *a , const MKL_INT
*ia , const MKL_INT *ja , const MKL_INT *desca , double *d , double *e , MKL_Complex16
*tau , MKL_Complex16 *work , const MKL_INT *lwork , MKL_INT *info );

void pslaed0 (const MKL_INT *n , float *d , float *e , float *q , const MKL_INT *iq ,
const MKL_INT *jq , const MKL_INT *descq , float *work , MKL_INT *iwork , MKL_INT
*info );
```

```

void pdlaed0 (const MKL_INT *n , double *d , double *e , double *q , const MKL_INT
*iq , const MKL_INT *jq , const MKL_INT *descq , double *work , MKL_INT *iwork ,
MKL_INT *info );

void pslaed1 (const MKL_INT *n , const MKL_INT *n1 , float *d , const MKL_INT *id ,
float *q , const MKL_INT *iq , const MKL_INT *jq , const MKL_INT *descq , const float
*rho , float *work , MKL_INT *iwork , MKL_INT *info );

void pdlaed1 (const MKL_INT *n , const MKL_INT *n1 , double *d , const MKL_INT *id ,
double *q , const MKL_INT *iq , const MKL_INT *jq , const MKL_INT *descq , const double
*rho , double *work , MKL_INT *iwork , MKL_INT *info );

void pslaed2 (const MKL_INT *ictxt , MKL_INT *k , const MKL_INT *n , const MKL_INT
*n1 , const MKL_INT *nb , float *d , const MKL_INT *drow , const MKL_INT *dcol , float
*q , const MKL_INT *ldq , float *rho , const float *z , float *w , float *dlamda , float
*q2 , const MKL_INT *ldq2 , float *qbuf , MKL_INT *ctot , MKL_INT *psm , const MKL_INT
*npcol , MKL_INT *indx , MKL_INT *indxc , MKL_INT *indxp , MKL_INT *indcol , MKL_INT
*coltyp , MKL_INT *nn , MKL_INT *nn1 , MKL_INT *nn2 , MKL_INT *ib1 , MKL_INT *ib2 );

void pdlaed2 (const MKL_INT *ictxt , MKL_INT *k , const MKL_INT *n , const MKL_INT
*n1 , const MKL_INT *nb , double *d , const MKL_INT *drow , const MKL_INT *dcol ,
double *q , const MKL_INT *ldq , double *rho , const double *z , double *w , double
*dlamda , double *q2 , const MKL_INT *ldq2 , double *qbuf , MKL_INT *ctot , MKL_INT
*psm , const MKL_INT *npcol , MKL_INT *indx , MKL_INT *indxc , MKL_INT *indxp , MKL_INT
*indcol , MKL_INT *coltyp , MKL_INT *nn , MKL_INT *nn1 , MKL_INT *nn2 , MKL_INT *ib1 ,
MKL_INT *ib2 );

void pslaed3 (const MKL_INT *ictxt , MKL_INT *k , const MKL_INT *n , const MKL_INT
*nb , float *d , const MKL_INT *drow , const MKL_INT *dcol , float *rho , float
*dlamda , float *w , const float *z , float *u , const MKL_INT *ldu , float *buf ,
MKL_INT *indx , MKL_INT *indcol , MKL_INT *indrow , MKL_INT *indxr , MKL_INT *indxc ,
MKL_INT *ctot , const MKL_INT *npcol , MKL_INT *info );

void pdlaed3 (const MKL_INT *ictxt , MKL_INT *k , const MKL_INT *n , const MKL_INT
*nb , double *d , const MKL_INT *drow , const MKL_INT *dcol , double *rho , double
*dlamda , double *w , const double *z , double *u , const MKL_INT *ldu , double *buf ,
MKL_INT *indx , MKL_INT *indcol , MKL_INT *indrow , MKL_INT *indxr , MKL_INT *indxc ,
MKL_INT *ctot , const MKL_INT *npcol , MKL_INT *info );

void pslaedz (const MKL_INT *n , const MKL_INT *n1 , const MKL_INT *id , const float
*q , const MKL_INT *iq , const MKL_INT *jq , const MKL_INT *ldq , const MKL_INT
*descq , float *z , float *work );

void pdlaedz (const MKL_INT *n , const MKL_INT *n1 , const MKL_INT *id , const double
*q , const MKL_INT *iq , const MKL_INT *jq , const MKL_INT *ldq , const MKL_INT
*descq , double *z , double *work );

void pdlaiectb (const double *sigma , const MKL_INT *n , const double *d , MKL_INT
*count );

void pdlaiectl (const double *sigma , const MKL_INT *n , const double *d , MKL_INT
*count );

void slamov (const char *UPLO , const MKL_INT *M , const MKL_INT *N , const float *A ,
const MKL_INT *LDA , float *B , const MKL_INT *LDB );

void dlamov (const char *UPLO , const MKL_INT *M , const MKL_INT *N , const double *A ,
const MKL_INT *LDA , double *B , const MKL_INT *LDB );

void clamov (const char *UPLO , const MKL_INT *M , const MKL_INT *N , const
MKL_Complex8 *A , const MKL_INT *LDA , MKL_Complex8 *B , const MKL_INT *LDB );

void zlamov (const char *UPLO , const MKL_INT *M , const MKL_INT *N , const
MKL_Complex16 *A , const MKL_INT *LDA , MKL_Complex16 *B , const MKL_INT *LDB );

```

```

void pslamrld (const MKL_INT *n , float *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , float *b , const MKL_INT *ib , const MKL_INT *jb , const MKL_INT
*descb );

void pdlamrld (const MKL_INT *n , double *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , double *b , const MKL_INT *ib , const MKL_INT *jb , const
MKL_INT *descb );

void pclamrld (const MKL_INT *n , MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT
*ja , const MKL_INT *desca , MKL_Complex8 *b , const MKL_INT *ib , const MKL_INT *jb ,
const MKL_INT *descb );

void pzlamrld (const MKL_INT *n , MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT
*ja , const MKL_INT *desca , MKL_Complex16 *b , const MKL_INT *ib , const MKL_INT *jb ,
const MKL_INT *descb );

void clanv2 (MKL_Complex8 *a , MKL_Complex8 *b , MKL_Complex8 *c , MKL_Complex8 *d ,
MKL_Complex8 *rt1 , MKL_Complex8 *rt2 , float *cs , MKL_Complex8 *sn );

void zlanv2 (MKL_Complex16 *a , MKL_Complex16 *b , MKL_Complex16 *c , MKL_Complex16
*d , MKL_Complex16 *rt1 , MKL_Complex16 *rt2 , double *cs , MKL_Complex16 *sn );

void pclattrs (const char *uplo , const char *trans , const char *diag , const char
*normin , const MKL_INT *n , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT
*ja , const MKL_INT *desca , MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , float *scale , float *cnorm , MKL_INT *info );

void pzlattrs (const char *uplo , const char *trans , const char *diag , const char
*normin , const MKL_INT *n , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT
*ja , const MKL_INT *desca , MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , double *scale , double *cnorm , MKL_INT *info );

void pssytrd (const char *uplo , const MKL_INT *n , float *a , const MKL_INT *ia ,
const MKL_INT *ja , const MKL_INT *desca , float *d , float *e , float *tau , float
*work , const MKL_INT *lwork , MKL_INT *info );

void pdsytrd (const char *uplo , const MKL_INT *n , double *a , const MKL_INT *ia ,
const MKL_INT *ja , const MKL_INT *desca , double *d , double *e , double *tau , double
*work , const MKL_INT *lwork , MKL_INT *info );

MKL_INT piparmq (const MKL_INT *ictxt , const MKL_INT *ispec , const char *name , const
char *opts , const MKL_INT *n , const MKL_INT *ilo , const MKL_INT *ihi , const MKL_INT
*lworknb );

```

For descriptions of these functions, please see <http://www.netlib.org/scalapack/explore-html/files.html>.

ScaLAPACK Utility Functions and Routines

This section describes ScaLAPACK utility functions and routines. Summary information about these routines is given in the following table:

ScaLAPACK Utility Functions and Routines

Routine Name	Data Types	Description
p?labad	s,d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
p?lachieee	s,d	Performs a simple check for the features of the IEEE standard.
p?lamch	s,d	Determines machine parameters for floating-point arithmetic.
p?lasnbt	s,d	Computes the position of the sign bit of a floating-point number.
descinit	N/A	Initializes the array descriptor for distributed matrix.

Routine Name	Data Types	Description
<code>numroc</code>	N/A	Computes the number of rows or columns of a distributed matrix owned by the process.

See Also

`pxerbla` Error handling routine called by ScaLAPACK routines.

p?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
void pslabad (MKL_INT *ictxt , float *small , float *large );
void pdlabad (MKL_INT *ictxt , double *small , double *large );
```

Include Files

- `mkl_scalapack.h`

Description

The `p?labad` function takes as input the values computed by `p?lamch` for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This function is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by `p?lamch`. This function is needed because `p?lamch` does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

In addition, this function performs a global minimization and maximization on these values, to support heterogeneous computing networks.

Input Parameters

<i>ictxt</i>	(global) The BLACS context handle in which the computation takes place.
<i>small</i>	(local). On entry, the underflow threshold as computed by <code>p?lamch</code> .
<i>large</i>	(local). On entry, the overflow threshold as computed by <code>p?lamch</code> .

Output Parameters

<i>small</i>	(local). On exit, if $\log_{10}(\textit{large})$ is sufficiently large, the square root of <i>small</i> , otherwise unchanged.
<i>large</i>	(local). On exit, if $\log_{10}(\textit{large})$ is sufficiently large, the square root of <i>large</i> , otherwise unchanged.

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?latchieee

Performs a simple check for the features of the IEEE standard.

Syntax

```
void pslatchieee (MKL_INT *isieee , float *rmax , float *rmin );
void pdlatchieee (MKL_INT *isieee , float *rmax , float *rmin );
```

Include Files

- mkl_scalapack.h

Description

The p?latchieee function performs a simple check to make sure that the features of the IEEE standard are implemented. In some implementations, p?latchieee may not return.

This is a ScaLAPACK internal function and arguments are not checked for unreasonable values.

Input Parameters

<i>rmax</i>	(local). The overflow threshold (= ?lamch ('O')).
<i>rmin</i>	(local). The underflow threshold (= ?lamch ('U')).

Output Parameters

<i>isieee</i>	(local). On exit, <i>isieee</i> = 1 implies that all the features of the IEEE standard that we rely on are implemented. On exit, <i>isieee</i> = 0 implies that some the features of the IEEE standard that we rely on are missing.
---------------	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

```
float pslamch (MKL_INT *ictxt , char *cmach );
double pdlamch (MKL_INT *ictxt , char *cmach );
```

Include Files

- mkl_scalapack.h

Description

The p?lamch function determines single precision machine parameters.

Input Parameters

<i>ictxt</i>	(global). The BLACS context handle in which the computation takes place.
<i>cmach</i>	<p>(global)</p> <p>Specifies the value to be returned by <code>p?lamch</code>:</p> <p>= 'E' or 'e', <code>p?lamch := eps</code></p> <p>= 'S' or 's', <code>p?lamch := sfmin</code></p> <p>= 'B' or 'b', <code>p?lamch := base</code></p> <p>= 'P' or 'p', <code>p?lamch := eps*base</code></p> <p>= 'N' or 'n', <code>p?lamch := t</code></p> <p>= 'R' or 'r', <code>p?lamch := rnd</code></p> <p>= 'M' or 'm', <code>p?lamch := emin</code></p> <p>= 'U' or 'u', <code>p?lamch := rmin</code></p> <p>= 'L' or 'l', <code>p?lamch := emax</code></p> <p>= 'O' or 'o', <code>p?lamch := rmax</code>,</p> <p>where</p> <p><code>eps</code> = relative machine precision</p> <p><code>sfmin</code> = safe minimum, such that <code>1/sfmin</code> does not overflow</p> <p><code>base</code> = base of the machine</p> <p><code>prec</code> = <code>eps*base</code></p> <p><code>t</code> = number of (base) digits in the mantissa</p> <p><code>rnd</code> = 1.0 when rounding occurs in addition, 0.0 otherwise</p> <p><code>emin</code> = minimum exponent before (gradual) underflow</p> <p><code>rmin</code> = underflow threshold - $\text{base}^{(\text{emin}-1)}$</p> <p><code>emax</code> = largest exponent before overflow</p> <p><code>rmax</code> = overflow threshold - $(\text{base}^{\text{emax}}) * (1-\text{eps})$</p>

Output Parameters

<i>val</i>	Value returned by the function.
------------	---------------------------------

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

`p?lasnbt`

Computes the position of the sign bit of a floating-point number.

Syntax

```
void pslasbnt (MKL_INT *ieflag );
void pdlasbnt (MKL_INT *ieflag );
```


Include Files

- `mkl_scalapack.h`

Description

The `p?lasnbt` function finds the position of the signbit of a single/double precision floating point number. This function assumes IEEE arithmetic, and hence, tests only the 32-nd bit (for single precision) or 32-nd and 64-th bits (for double precision) as a possibility for the signbit. `sizeof(int)` is assumed equal to 4 bytes.

If a compile time flag (`NO_IEEE`) indicates that the machine does not have IEEE arithmetic, `ieflag = 0` is returned.

Output Parameters

`ieflag` This flag indicates the position of the signbit of any single/double precision floating point number.

`ieflag = 0`, if the compile time flag `NO_IEEE` indicates that the machine does not have IEEE arithmetic, or if `sizeof(int)` is different from 4 bytes.

`ieflag = 1` indicates that the signbit is the 32-nd bit for a single precision function.

In the case of a double precision function:

`ieflag = 1` indicates that the signbit is the 32-nd bit (Big Endian).

`ieflag = 2` indicates that the signbit is the 64-th bit (Little Endian).

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

descinit

Initializes the array descriptor for distributed matrix.

Syntax

```
void descinit (MKL_INT *desc, const MKL_INT *m, const MKL_INT *n, const MKL_INT *mb,
const MKL_INT *nb, const MKL_INT *irsrc, const MKL_INT *icsrc, const MKL_INT *ictxt,
const MKL_INT *lld, MKL_INT *info);
```

Description

The `descint` function initializes the array descriptor for distributed matrix.

Input Parameters

<code>desc</code>	(global) array of dimension <code>DLEN_</code> . The array descriptor of a distributed matrix to be set.
<code>m</code>	(global input) The number of rows in the distributed matrix. $M \geq 0$.
<code>n</code>	(global input) The number of columns in the distributed matrix. $N \geq 0$.
<code>mb</code>	(global input) The blocking factor used to distribute the rows of the matrix. $MB \geq 1$.
<code>nb</code>	(global input) The blocking factor used to distribute the columns of the matrix. $NB \geq 1$.

<i>lrsrc</i>	(global input) The process row over which the first row of the matrix is distributed. $0 \leq \text{IRSRC} < \text{NPROW}$.
<i>lcsrc</i>	(global input) The process column over which the first column of the matrix is distributed. $0 \leq \text{ICSRC} < \text{NPCOL}$.
<i>ictxt</i>	(global input) The BLACS context handle, indicating the global context of the operation on the matrix. The context itself is global.
<i>lld</i>	(local input) The leading dimension of the local array storing the local blocks of the distributed matrix. $\text{LLD} \geq \text{MAX}(1, \text{LOCr}(\text{M}))$. $\text{LOCr}()$ denotes the number of rows of a global dense matrix that the process in a grid receives after data distributing.

Output Parameters

<i>info</i>	(output)
	= 0: successful exit
	< 0: if $\text{INFO} = -i$, the i -th argument had an illegal value

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

numroc

Computes the number of rows or columns of a distributed matrix owned by the process.

Syntax

```
MKL_INT numroc (const MKL_INT *n, const MKL_INT *nb, const MKL_INT *iproc, const
MKL_INT *srcproc, const MKL_INT *nprocs);
```

Description

The `numroc` function computes the number of rows or columns of a distributed matrix owned by the process.

Input Parameters

<i>n</i>	(global input) The number of rows/columns in distributed matrix.
<i>nb</i>	(global input) Block size, size of the blocks the distributed matrix is split into.
<i>iproc</i>	(local input) The coordinate of the process whose local array row or column is to be determined.
<i>srcproc</i>	(global input) The coordinate of the process that possesses the first row or column of the distributed matrix.
<i>nprocs</i>	(global input) The total number processes over which the matrix is distributed.

Output Parameters

<i>info</i>	(output) Value returned by the function.
-------------	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

ScaLAPACK Redistribution/Copy Routines

This section describes ScaLAPACK redistribution/copy routines. Summary information about these routines is given in the following table:

ScaLAPACK Redistribution/Copy Routines

Routine Name	Data Types	Description
p?gemr2d	s,d,c,z,i	Copies a submatrix from one general rectangular matrix to another.
p?trmr2d	s,d,c,z,i	Copies a submatrix from one trapezoidal matrix to another.

See Also

[pxerbla](#) Error handling routine called by ScaLAPACK routines.

[p?gemr2d](#)

Copies a submatrix from one general rectangular matrix to another.

Syntax

```

void psgemr2d (MKL_INT *m, MKL_INT *n, float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );

void pdgemr2d (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );

void pcgemr2d (MKL_INT *m , MKL_INT *n MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT
*ictxt );

void pzgemr2d (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_INT *ictxt );

void pigemr2d (MKL_INT *m , MKL_INT *n , MKL_INT *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT
*ictxt );

```

Include Files

- `mkc_scalapack.h`

Description

The `p?gemr2d` function copies the indicated matrix or submatrix of *A* to the indicated matrix or submatrix of *B*. It provides a truly general copy from any block cyclicly-distributed matrix or submatrix to any other block cyclicly-distributed matrix or submatrix. With `p?trmr2d`, these functions are the only ones in the ScaLAPACK library which provide inter-context operations: they can take a matrix or submatrix *A* in context *A* (distributed over process grid *A*) and copy it to a matrix or submatrix *B* in context *B* (distributed over process grid *B*).

There does not need to be a relationship between the two operand matrices or submatrices other than their global size and the fact that they are both legal block cyclicly-distributed matrices or submatrices. This means that they can, for example, be distributed across different process grids, have varying block sizes and differing matrix starting points, or be contained in different sized distributed matrices.

Take care when context *A* is disjoint from context *B*. The general rules for which parameters need to be set are:

- All calling processes must have the correct *m* and *n*.
- Processes in context *A* must correctly define all parameters describing *A*.
- Processes in context *B* must correctly define all parameters describing *B*.
- Processes which are not members of context *A* must pass *ctxt_a* = -1 and need not set other parameters describing *A*.
- Processes which are not members of context *B* must pass *ctxt_b* = -1 and need not set other parameters describing *B*.

Because of its generality, `p?gemr2d` can be used for many operations not usually associated with copy functions. For instance, it can be used to take a matrix on one process and distribute it across a process grid, or the reverse. If a supercomputer is grouped into a virtual parallel machine with a workstation, for instance, this function can be used to move the matrix from the workstation to the supercomputer and back. In ScaLAPACK, it is called to copy matrices from a two-dimensional process grid to a one-dimensional process grid. It can be used to redistribute matrices so that distributions providing maximal performance can be used by various component libraries, as well.

Note that this function requires an array descriptor with *dtype_* = 1.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>m</i>	(global) The number of rows of matrix <i>A</i> to be copied ($m \geq 0$).
<i>n</i>	(global) The number of columns of matrix <i>A</i> to be copied ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to array of size <code>lld_a * LOcc(ja+n-1)</code> containing the source matrix <i>A</i> .
<i>ia, ja</i>	(global) The row and column indices in the array <i>A</i> indicating the first row and the first column, respectively, of the submatrix of <i>A</i> to copy. $1 \leq ia \leq total_rows_in_a - m + 1$, $1 \leq ja \leq total_columns_in_a - n + 1$.
<i>desca</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>A</i> . Only <i>dtype_a</i> = 1 is supported, so <i>dlen_</i> = 9. If the calling process is not part of the context of <i>A</i> , <i>ctxt_a</i> must be equal to -1.
<i>ib, jb</i>	(global) The row and column indices in the array <i>B</i> indicating the first row and the first column, respectively, of the submatrix <i>B</i> to which to copy the matrix. $1 \leq ib \leq total_rows_in_b - m + 1$, $1 \leq jb \leq total_columns_in_b - n + 1$.
<i>descb</i>	(global and local) array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> . Only <i>dtype_b</i> = 1 is supported, so <i>dlen_</i> = 9.

If the calling process is not part of the context of B , $ctxt_b$ must be equal to -1.

$ictxt$

(global).

The context encompassing at least the union of all processes in context A and context B . All processes in the context $ictxt$ must call this function, even if they do not own a piece of either matrix.

Output Parameters

b

Pointer into the local memory to array of size $lld_b * LOCC(jb+n-1)$.

Overwritten by the submatrix from A .

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

p?trmr2d

Copies a submatrix from one trapezoidal matrix to another.

Syntax

```
void pstrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *ictxt );

void pdtrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );

void pctrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );

void pztrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );

void pitrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , MKL_INT *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *ictxt );
```

Include Files

- mkl_scalapack.h

Description

The `p?trmr2d` function copies the indicated matrix or submatrix of A to the indicated matrix or submatrix of B . It provides a truly general copy from any block cyclicly-distributed matrix or submatrix to any other block cyclicly-distributed matrix or submatrix. With `p?gemr2d`, these functions are the only ones in the ScaLAPACK library which provide inter-context operations: they can take a matrix or submatrix A in context A (distributed over process grid A) and copy it to a matrix or submatrix B in context B (distributed over process grid B).

The `p?trmr2d` function assumes the matrix or submatrix to be trapezoidal. Only the upper or lower part is copied, and the other part is unchanged.

There does not need to be a relationship between the two operand matrices or submatrices other than their global size and the fact that they are both legal block cyclicly-distributed matrices or submatrices. This means that they can, for example, be distributed across different process grids, have varying block sizes and differing matrix starting points, or be contained in different sized distributed matrices.

Take care when context *A* is disjoint from context *B*. The general rules for which parameters need to be set are:

- All calling processes must have the correct *m* and *n*.
- Processes in context *A* must correctly define all parameters describing *A*.
- Processes in context *B* must correctly define all parameters describing *B*.
- Processes which are not members of context *A* must pass *ctxt_a* = -1 and need not set other parameters describing *A*.
- Processes which are not members of context *B* must pass *ctxt_b* = -1 and need not set other parameters describing *B*.

Because of its generality, `p?trmr2d` can be used for many operations not usually associated with copy functions. For instance, it can be used to take a matrix on one process and distribute it across a process grid, or the reverse. If a supercomputer is grouped into a virtual parallel machine with a workstation, for instance, this function can be used to move the matrix from the workstation to the supercomputer and back. In ScaLAPACK, it is called to copy matrices from a two-dimensional process grid to a one-dimensional process grid. It can be used to redistribute matrices so that distributions providing maximal performance can be used by various component libraries, as well.

Note that this function requires an array descriptor with *dtype_* = 1.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>uplo</i>	(global) Specifies whether to copy the upper or lower part of the matrix or submatrix.
<i>uplo</i> = 'U'	Copy the upper triangular part.
<i>uplo</i> = 'L'	Copy the lower triangular part.
<i>diag</i>	(global) Specifies whether to copy the diagonal of the matrix or submatrix.
<i>diag</i> = 'U'	Do not copy the diagonal.
<i>diag</i> = 'N'	Copy the diagonal.
<i>m</i>	(global) The number of rows of matrix <i>A</i> to be copied ($m \geq 0$).
<i>n</i>	(global) The number of columns of matrix <i>A</i> to be copied ($n \geq 0$).
<i>a</i>	(local) Pointer into the local memory to array of size $lld_a * LOCC(ja+n-1)$ containing the source matrix <i>A</i> .
<i>ia, ja</i>	(global) The row and column indices in the array <i>A</i> indicating the first row and the first column, respectively, of the submatrix of <i>A</i> to copy. $1 \leq ia \leq total_rows_in_a - m + 1$, $1 \leq ja \leq total_columns_in_a - n + 1$.

<i>desca</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>A</i>.</p> <p>Only <i>dtype_a</i> = 1 is supported, so <i>dlen_</i> = 9.</p> <p>If the calling process is not part of the context of <i>A</i>, <i>ctxt_a</i> must be equal to -1.</p>
<i>ib, jb</i>	<p>(global) The row and column indices in the array <i>B</i> indicating the first row and the first column, respectively, of the submatrix <i>B</i> to which to copy the matrix. $1 \leq ib \leq total_rows_in_b - m + 1$, $1 \leq jb \leq total_columns_in_b - n + 1$.</p>
<i>descb</i>	<p>(global and local) array of size <i>dlen_</i>. The array descriptor for the distributed matrix <i>B</i>.</p> <p>Only <i>dtype_b</i> = 1 is supported, so <i>dlen_</i> = 9.</p> <p>If the calling process is not part of the context of <i>B</i>, <i>ctxt_b</i> must be equal to -1.</p>
<i>ictxt</i>	<p>(global).</p> <p>The context encompassing at least the union of all processes in context <i>A</i> and context <i>B</i>. All processes in the context <i>ictxt</i> must call this function, even if they do not own a piece of either matrix.</p>

Output Parameters

<i>b</i>	<p>Pointer into the local memory to array of size <i>lld_b</i>*<i>LOCc</i>(<i>jb</i>+<i>n</i>-1).</p> <p>Overwritten by the submatrix from <i>A</i>.</p>
----------	--

See Also

[Overview](#) for details of ScaLAPACK array descriptor structures and related notations.

Sparse Solver Routines

Intel® oneAPI Math Kernel Library (oneMKL) sparse solver algorithms for solving real or complex, symmetric, structurally symmetric or nonsymmetric, positive definite, indefinite or Hermitian square sparse linear system of algebraic equations.

The terms and concepts required to understand the use of the Intel® oneAPI Math Kernel Library (oneMKL) sparse solver routines are discussed in the [Appendix "Linear Solvers Basics"](#). If you are familiar with linear sparse solvers and sparse matrix storage schemes, you can skip these sections and go directly to the interface descriptions.

See the description of

- the direct sparse solver based on PARDISO*, which is referred to here as [Intel MKL PARDISO](#);
- the alternative interface for the direct sparse solver, which is referred to here as the [DSS interface](#);
- [iterative sparse solvers \(ISS\)](#) based on the reverse communication interface (RCI);
- [preconditioners](#) based on the incomplete LU factorization technique.
- a direct sparse [solver](#) based on QR decomposition.

oneMKL PARDISO - Parallel Direct Sparse Solver Interface

This section describes the interface to the shared-memory multiprocessing parallel direct sparse solver known as the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver.

The Intel® oneAPI Math Kernel Library (oneMKL) PARDISO package is a high-performance, robust, memory efficient, and easy to use software package for solving large sparse linear systems of equations on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode

techniques [Schenk00-2]. To improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining parallelism is used with a combination of left- and right-looking supernode techniques [Schenk00, Schenk01, Schenk02, Schenk03]. The parallel pivoting methods allow complete supernode pivoting to compromise numerical stability and scalability during the factorization process. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture.

Product and Performance Information

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

Notice revision #20201201

The following table lists the names of the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO routines and describes their general use.

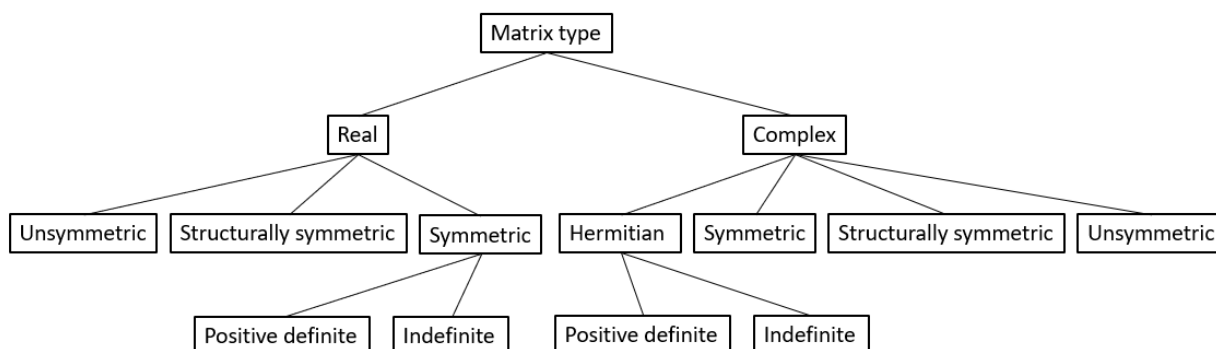
oneMKL PARDISO Routines

Routine	Description
<code>pardisoinit</code>	Initializes Intel® oneAPI Math Kernel Library (oneMKL) PARDISO with default parameters depending on the matrix type.
<code>pardiso</code>	Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.
<code>pardiso_64</code>	Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides, 64-bit integer version.
<code>mkl_pardiso_pivot</code>	Replaces routine which handles Intel® oneAPI Math Kernel Library (oneMKL) PARDISO pivots with user-defined routine.
<code>pardiso_getdiag</code>	Returns diagonal elements of initial and factorized matrix.
<code>pardiso_export</code>	Places pointers dedicated for sparse representation of requested matrix into MKL PARDISO.
<code>pardiso_handle_store</code>	Store internal structures from <code>pardiso</code> to a file.
<code>pardiso_handle_restore</code>	Restore <code>pardiso</code> internal structures from a file.
<code>pardiso_handle_delete</code>	Delete files with <code>pardiso</code> internal structure data.
<code>pardiso_handle_store_64</code>	Store internal structures from <code>pardiso_64</code> to a file.
<code>pardiso_handle_restore_64</code>	Restore <code>pardiso_64</code> internal structures from a file.
<code>pardiso_handle_delete_64</code>	Delete files with <code>pardiso_64</code> internal structure data.

The Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver supports a wide range of real and complex sparse matrix types (see the figure below).

[__border__top](#)

Sparse Matrices That Can Be Solved with the oneMKL PARDISO Solver



The Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver performs four tasks:

- analysis and symbolic factorization
- numerical factorization
- forward and backward substitution including iterative refinement
- termination to release all internal solver memory.

To find code examples that use Intel® oneAPI Math Kernel Library (oneMKL) PARDISO routines to solve systems of linear equations, unzip theC archive file in the `examples` folder of the Intel® oneAPI Math Kernel Library (oneMKL) installation directory. Code examples will be in the `examples/solverc/source` folder.

Supported Matrix Types

The analysis steps performed by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO depend on the structure of the input matrix A .

Symmetric Matrices

The solver first computes a symmetric fill-in reducing permutation P based on either the minimum degree algorithm [Liu85] or the nested dissection algorithm from the METIS package [Karypis98] (both included with Intel® oneAPI Math Kernel Library (oneMKL)), followed by the parallel left-right looking numerical Cholesky factorization [Schenk00-2] of $PAP^T = LL^T$ for symmetric positive-definite matrices, or $PAP^T = LDL^T$ for symmetric indefinite matrices. The solver uses diagonal pivoting, or 1x1 and 2x2 Bunch-Kaufman pivoting for symmetric indefinite matrices. An approximation of X is found by forward and backward substitution and optional iterative refinement.

Whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal supernode block, the coefficient matrix is perturbed. One or two passes of iterative refinement may be required to correct the effect of the perturbations. This restricting notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. Furthermore, for a large set of matrices from different applications areas, this method is as accurate as a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Another method of improving the pivoting accuracy is to use symmetric weighted matching algorithms. These algorithms identify large entries in the coefficient matrix A that, if permuted close to the diagonal, permit the factorization process

to identify more acceptable pivots and proceed with fewer pivot perturbations. These algorithms are based on maximum weighted matchings and improve the quality of the factor in a complementary way to the alternative of using more complete pivoting techniques.

The inertia is also computed for real symmetric indefinite matrices.

Structurally Symmetric Matrices

The solver first computes a symmetric fill-in reducing permutation P followed by the parallel numerical factorization of $PAP^T = QLU^T$. The solver uses partial pivoting in the supernodes and an approximation of X is found by forward and backward substitution and optional iterative refinement.

Nonsymmetric Matrices

The solver first computes a nonsymmetric permutation P_{MPS} and scaling matrices D_r and D_c with the aim of placing large entries on the diagonal to enhance reliability of the numerical factorization process [Duff99]. In the next step the solver computes a fill-in reducing permutation P based on the matrix $P_{MPS}A + (P_{MPS}A)^T$ followed by the parallel numerical factorization

$$QLUR = PP_{MPS}D_rAD_cP$$

with supernode pivoting matrices Q and R . When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99]. The magnitude of the potential pivot is tested against a constant threshold of

$$\alpha = \text{eps} * ||A2||_{\text{inf}},$$

where eps is the machine precision, $A2 = P * P_{MPS} * D_r * A * D_c * P$, and $||A2||_{\text{inf}}$ is the infinity norm of A . Any tiny pivots encountered during elimination are set to the sign $(l_{II}) * \text{eps} * ||A2||_{\text{inf}}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice the diagonal elements are rarely modified for a large class of matrices. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed.

Sparse Data Storage

Intel® oneAPI Math Kernel Library (oneMKL) PARDISO stores sparse data in several formats:

- CSR3: The 3-array variation of the compressed sparse row format described in [Three Array Variation of CSR Format](#).
- BSR3: The three-array variation of the block compressed sparse row format described in [Three Array Variation of BSR Format](#). Use `iparm[36]` to specify the block size.
- VBSR: Variable BSR format. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO analyzes the matrix provided in CSR3 format and converts it into an internal structure which can improve performance for matrices with a block structure. Use `iparm[36] = -t` ($0 < t \leq 100$) to specify use of internal VBSR format and to set the degree of similarity required to combine elements of the matrix. For example, if you set `iparm[36] = -80`, two rows of the input matrix are combined when their non-zero patterns are 80% or more similar.

NOTE

Intel® oneAPI Math Kernel Library (oneMKL) supports only the VBSR format for real and symmetric positive definite or indefinite matrices (*mtype* = 2 or *mtype* = -2).

Intel® oneAPI Math Kernel Library (oneMKL) supports these features for all matrix types as long as *asiparm*[23]=1:

- *iparm*[30] > 0: Partial solution
- *iparm*[35] > 0: Schur complement
- *iparm*[59] > 0: OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO

For all storage formats, the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO parameter *ja* is used for the *columns* array, *ia* is used for *rowIndex*, and *a* is used for *values*. The algorithms in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO require column indices *ja* to be in increasing order per row and that the diagonal element in each row be present for any structurally symmetric matrix. For symmetric or nonsymmetric matrices the diagonal elements which are equal to zero are not necessary.

Caution

Intel® oneAPI Math Kernel Library (oneMKL) PARDISO column indices *ja* must be in increasing order per row. You can validate the sparse matrix structure with the matrix checker (*iparm*[26])

NOTE

While the presence of zero diagonal elements for symmetric matrices is not required, you should explicitly set zero diagonal elements for symmetric matrices. Otherwise, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO creates internal copies of arrays *ia*, *ja*, and *a* full of diagonal elements, which require additional memory and computational time. However, the memory and time required the diagonal elements in internal arrays is usually not significant compared to the memory and the time required to factor and solve the matrix.

Product and Performance Information

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

Notice revision #20201201

Storage of Matrices

By default, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO stores data in RAM. This is referred to as In-Core (IC) mode. However, you can specify that Intel® oneAPI Math Kernel Library (oneMKL) PARDISO store matrices on disk by setting *iparm*[59]. This mode is called the Out-of-Core (OOC) mode.

You can set the following parameters for the OOC mode.

Parameter/Environment Variable Name	Description
MKL_PARDISO_OOC_PATH	Directory for storing data created in the OOC mode.
MKL_PARDISO_OOC_FILE_NAME	Full file name (incl. path) which will be used for the OOC files

Parameter/Environment Variable Name	Description
MKL_PARDISO_OOC_MAX_CORE_SIZE	Maximum size of RAM (in megabytes) available for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO
MKL_PARDISO_OOC_MAX_SWAP_SIZE	Maximum swap size (in megabytes) available for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO
MKL_PARDISO_OOC_KEEP_FILE	A flag which determines whether temporary data files will be deleted or stored

By default, the current working directory is used in the OOC mode as a directory path for storing data. All work arrays will be stored in files named `ooc_temp` with different extensions. When `MKL_PARDISO_OOC_FILE_NAME` is not set and `MKL_PARDISO_OOC_PATH` is set, the names for the created files will contain `<path>/mkl_pardiso` or `<path>\mkl_pardiso` depending on the OS. Setting `MKL_PARDISO_OOC_FILE_NAME=<filename>` will override the path which could have been set in `MKL_PARDISO_OOC_PATH`. In this case `<filename>` will be used for naming the OOC files.

By default, `MKL_PARDISO_OOC_MAX_CORE_SIZE` is 2000 (MB) and `MKL_PARDISO_OOC_MAX_SWAP_SIZE` is 0.

NOTE

Do not set the sum of `MKL_PARDISO_OOC_MAX_CORE_SIZE` and `MKL_PARDISO_OOC_MAX_SWAP_SIZE` greater than the size of the RAM plus the size of the swap memory. Be sure to allow enough free memory for the operating system and any other processes which need to be running.

By default, all temporary data files will be deleted. For keeping them it is required to set `MKL_PARDISO_OOC_KEEP_FILE` to 0.

OOC parameters can be set in a configuration file. You can set the path to this file and its name using environmental variables `MKL_PARDISO_OOC_CFG_PATH` and `MKL_PARDISO_OOC_CFG_FILE_NAME`.

For setting parameters of OOC mode either environment variables or a configuration file can be used. When the last option is chosen, by default the name of the file is `pardiso_ooc.cfg` and it should be placed in the working directory. If needed, the user can set the path to the configuration file using environmental variables `MKL_PARDISO_OOC_CFG_PATH` and `MKL_PARDISO_OOC_CFG_FILE_NAME`. These variables specify the path and filename as follows:

- Linux* OS and OS X*: `<MKL_PARDISO_OOC_CFG_PATH>/ <MKL_PARDISO_OOC_CFG_FILE_NAME>`
- Windows* OS: `<MKL_PARDISO_OOC_CFG_PATH>\<MKL_PARDISO_OOC_CFG_FILE_NAME>`

An example of the configuration file:

```
MKL_PARDISO_OOC_PATH = <path>
MKL_PARDISO_OOC_MAX_CORE_SIZE = N
MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

Caution

The maximum length of the path lines in the configuration files is 1000 characters.

Alternatively, the OOC parameters can be set as environment variables via command line.

For Linux* OS and OS X*:

```
export MKL_PARDISO_OOC_PATH = <path>
export MKL_PARDISO_OOC_MAX_CORE_SIZE = N
export MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
export MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

For Windows* OS:

```
set MKL_PARDISO_OOC_PATH = <path>
set MKL_PARDISO_OOC_MAX_CORE_SIZE = N
set MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
set MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

where <path> should follow the OS naming convention.

Direct-Iterative Preconditioning for Nonsymmetric Linear Systems

The solver uses a combination of direct and iterative methods [Sonn89] to accelerate the linear solution process for transient simulation. Most applications of sparse solvers require solutions of systems with gradually changing values of the nonzero coefficient matrix, but with an identical sparsity pattern. In these applications, the analysis phase of the solvers has to be performed only once and the numerical factorizations are the important time-consuming steps during the simulation. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses a numerical factorization and applies the factors in a preconditioned Krylov Subspace iteration. If the iteration does not converge, the solver automatically switches back to the numerical factorization. This method can be applied to nonsymmetric matrices in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. You can select the method using the `iparm[3]` input parameter. The `iparm[19]` parameter returns the error status after running Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

Single and Double Precision Computations

Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solves tasks using single or double precision. Each precision has its benefits and drawbacks. Double precision variables have more digits to store value, so the solver uses more memory for keeping data. But this mode solves matrices with better accuracy, which is especially important for input matrices with large condition numbers.

Single precision variables have fewer digits to store values, so the solver uses less memory than in the double precision mode. Additionally this mode usually takes less time. But as computations are made less precisely, only some systems of equations can be solved accurately enough using single precision.

Separate Forward and Backward Substitution

The solver execution step (see `parameterphase = 33` below) can be divided into two or three separate substitutions: forward, backward, and possible diagonal. This separation can be explained by the examples of solving systems with different matrix types.

A real symmetric positive definite matrix A (`mtype = 2`) is factored by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO as $A = L^*L^T$. In this case the solution of the system $A^*x=b$ can be found as sequence of substitutions: $L^*y=b$ (forward substitution, `phase = 331`) and $L^T*x=y$ (backward substitution, `phase = 333`).

A real nonsymmetric matrix A (`mtype = 11`) is factored by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO as $A = L^*U$. In this case the solution of the system $A^*x=b$ can be found by the following sequence: $L^*y=b$ (forward substitution, `phase = 331`) and $U^*x=y$ (backward substitution, `phase = 333`).

Solving a system with a real symmetric indefinite matrix A (`mtype = -2`) is slightly different from the cases above. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO factors this matrix as $A=LDL^T$, and the solution of the system $A^*x=b$ can be calculated as the following sequence of substitutions: $L^*y=b$ (forward

substitution, $phase = 331$), $D^*v=y$ (diagonal substitution, $phase = 332$), and finally $L^T*x=v$ (backward substitution, $phase = 333$). Diagonal substitution makes sense only for symmetric indefinite matrices ($mtype = -2, -4, 6$). For matrices of other types a solution can be found as described in the first two examples.

Caution

The number of refinement steps (`iparm[7]`) must be set to zero if a solution is calculated with separate substitutions ($phase = 331, 332, 333$), otherwise Intel® oneAPI Math Kernel Library (oneMKL) PARDISO produces the wrong result.

NOTE

Different pivoting (`iparm[20]`) produces different LDL^T factorization. Therefore results of forward, diagonal and backward substitutions with diagonal pivoting can differ from results of the same steps with Bunch-Kaufman pivoting. Of course, the final results of sequential execution of forward, diagonal and backward substitution are equal to the results of the full solving step ($phase=33$) regardless of the pivoting used.

Callback Function for Pivoting Control

In-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO allows you to control pivoting with a callback routine, `mkl_pardiso_pivot`. You can then use the `pardiso_getdiag` routine to access the diagonal elements. Set `iparm[55]` to 1 in order to use the callback functionality.

Low Rank Update

Use low rank update to accelerate the factorization step in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO when you use multiple matrices with identical structure and similar values. After calling `pardiso` in the usual manner for factorization ($phase = 12, 13, 22$, or 23) for some matrix *A1*, low rank update can be applied to the factorization step ($phase = 22$ or 23) of some matrix *A2* with identical structure.

To use the low rank update feature, set `iparm[38] = 1` while also setting `iparm[23] = 10`. Additionally, supply an array that lists the values in *A2* that are different from *A1* using the `perm` parameter as outlined in the `pardiso perm` parameter description.

Important

Low rank update can only be called for matrices with the exact same pattern of nonzero values. As such, the value of the `mtype`, `ia`, `ja`, and `iparm[23]` parameters should also be identical. In general, the low rank factorization should be called with the same parameters as the preceding factorization step for the same internal data structure handle (except for array *a*, `iparm[38]`, and `perm`).

Low rank update does not currently support Intel TBB threading. In this case, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO defaults to full factorization instead.

Low rank update cannot be used in combination with a user-supplied permutation vector - in other words, you must use the default values of `iparm[4] = 0`, `iparm[30] = 0`, and `iparm[35] = 0`. Additionally, `iparm[3]`, `iparm[5]`, `iparm[27]`, `iparm[36]`, `iparm[55]`, and `iparm[59]` must all be set to the default value of 0.

`pardiso`

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

```
void pardiso (_MKL_DSS_HANDLE_t pt, const MKL_INT *maxfct, const MKL_INT *mnum, const
MKL_INT *mtype, const MKL_INT *phase, const MKL_INT *n, const void *a, const MKL_INT
*ia, const MKL_INT *ja, MKL_INT *perm, const MKL_INT *nrhs, MKL_INT *iparm, const
MKL_INT *msglvl, void *b, void *x, MKL_INT *error);
```

Include Files

- mkl.h

Description

The `pardiso` routine calculates the solution of a set of sparse linear equations

$$A \cdot X = B$$

with single or multiple right-hand sides, using a parallel *LU*, *LDL*, or *LL^T* factorization, where *A* is an *n*-by-*n* matrix, and *X* and *B* are *n*-by-*nrhs* vectors or matrices.

Notes

- This routine supports usage of the `mkl_progress` with OpenMP, TBB, and sequential threading. See [mkl_progress](#) for details. The case of `iparm[23]=10` does not support this feature.
- If `iparm[26]` is set to 1 (Matrix checker), Intel® oneAPI Math Kernel Library PARDISO uses the auxiliary routine `sparse_matrix_checker` to check integer arrays `ia` and `ja`. `sparse_matrix_checker` has its own set of error values (from 21 to 24) that are returned in the event of an unsuccessful matrix check. For more details, refer to the `sparse_matrix_checker` documentation.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

pt

Array with size of 64.

Handle to internal data structure. The entries must be set to zero prior to the first call to `pardiso`. Unique for factorization.

Caution

After the first call to `pardiso` do not directly modify *pt*, as that could cause a serious memory leak.

Use the [pardiso_handle_store](#) or [pardiso_handle_store_64](#) routine to store the content of *pt* to a file. Restore the contents of *pt* from the file using [pardiso_handle_restore](#) or [pardiso_handle_restore_64](#). Use `pardiso_handle_store` and `pardiso_handle_restore` with `pardiso`, and `pardiso_handle_store_64` and `pardiso_handle_restore_64` with `pardiso_64`.

maxfct

Maximum number of factors with identical sparsity structure that must be kept in memory at the same time. In most applications this value is equal to 1. It is possible to store several different factorizations with the same nonzero structure at the same time in the internal data structure management of the solver.

pardiso can process several matrices with an identical matrix sparsity pattern and it can store the factors of these matrices at the same time. Matrices with a different sparsity structure can be kept in memory with different memory address pointers *pt*.

mnum

Indicates the actual matrix for the solution phase. With this scalar you can define which matrix to factorize. The value must be: $1 \leq mnum \leq maxfct$.

In most applications this value is 1.

mtype

Defines the matrix type, which influences the pivoting method. The Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver supports the following matrices:

1	real and structurally symmetric
2	real and symmetric positive definite
-2	real and symmetric indefinite
3	complex and structurally symmetric
4	complex and Hermitian positive definite
-4	complex and Hermitian indefinite
6	complex and symmetric
11	real and nonsymmetric
13	complex and nonsymmetric

phase

Controls the execution of the solver. Usually it is a two- or three-digit integer. The first digit indicates the starting phase of execution and the second digit indicates the ending phase. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including optional iterative refinement

This phase can be divided into two or three separate substitutions: forward, backward, and diagonal (see [Separate Forward and Backward Substitution](#)).

- Memory release phase (*phase*= 0 or *phase*= -1)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The *phase* parameter can have the following values:

<i>phase</i>	Solver Execution Steps
11	Analysis

<i>phase</i>	Solver Execution Steps
12	Analysis, numerical factorization
13	Analysis, numerical factorization, solve, iterative refinement
22	Numerical factorization
23	Numerical factorization, solve, iterative refinement
33	Solve, iterative refinement
331	like <i>phase</i> =33, but only forward substitution
332	like <i>phase</i> =33, but only diagonal substitution (if available)
333	like <i>phase</i> =33, but only backward substitution
0	Release internal memory for <i>L</i> and <i>U</i> matrix number <i>mnum</i>
-1	Release all internal memory for all matrices

If *iparm*[35] = 0, phases 331, 332, and 333 perform this decomposition:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{12} & L_{22} \end{bmatrix} \begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{21} \\ 0 & U_{22} \end{bmatrix}$$

If *iparm*[35] = 2, phases 331, 332, and 333 perform a different decomposition:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{12} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} U_{11} & U_{21} \\ 0 & I \end{bmatrix}$$

You can supply a custom implementation for phase 332 instead of calling `pardiso`. For example, it can be implemented with dense LAPACK functionality. Custom implementation also allows you to substitute the matrix *S* with your own.

NOTE

For very large Schur complement matrices use LAPACK functionality to compute the Schur complement vector instead of the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO phase 332 implementation.

n

Number of equations in the sparse linear systems of equations $A^*X = B$. Constraint: $n > 0$.

a

Array. Contains the non-zero elements of the coefficient matrix *A* corresponding to the indices in *ja*. The coefficient matrix can be either real or complex. The matrix must be stored in the three-array variant of the compressed sparse row (CSR3) or in the three-array variant of the block compressed sparse row (BSR3) format, and the matrix must be stored with increasing values of *ja* for each row.

For CSR3 format, the size of a is the same as that of ja . Refer to the *values* array description in [Three Array Variation of CSR Format](#) for more details.

For BSR3 format the size of a is the size of ja multiplied by the square of the block size. Refer to the *values* array description in [Three Array Variation of BSR Format](#) for more details.

NOTE

If you set `iparm[36]` to a negative value, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO converts the data from CSR3 format to an internal variable BSR (VBSR) format. See [Sparse Data Storage](#).

ia

Array, size $(n+1)$.

For CSR3 format, $ia[i]$ ($i < n$) points to the first column index of row i in the array ja . That is, $ia[i]$ gives the index of the element in array a that contains the first non-zero element from row i of A . The last element $ia[n]$ is taken to be equal to the number of non-zero elements in A , plus one. Refer to *rowIndex* array description in [Three Array Variation of CSR Format](#) for more details.

For BSR3 format, $ia[i]$ ($i < n$) points to the first column index of row i in the array ja . That is, $ia[i]$ gives the index of the element in array a that contains the first non-zero block from row i of A . The last element $ia[n]$ is taken to be equal to the number of non-zero blocks in A , plus one. Refer to *rowIndex* array description in [Three Array Variation of BSR Format](#) for more details.

The array ia is accessed in all phases of the solution process.

Indexing of ia is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter `iparm[34]`.

ja

For CSR3 format, array ja contains column indices of the sparse matrix A . It is important that the indices are in increasing order per row. For structurally symmetric matrices it is assumed that all diagonal elements are stored (even if they are zeros) in the list of non-zero elements in a and ja . For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in [Three Array Variation of CSR Format](#).

For BSR3 format, array ja contains column indices of the sparse matrix A . It is important that the indices are in increasing order per row. For structurally symmetric matrices it is assumed that all diagonal blocks are stored (even if they are zeros) in the list of non-zero blocks in a and ja . For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in [Three Array Variation of BSR Format](#).

The array ja is accessed in all phases of the solution process.

Indexing of ja is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter `iparm[34]`.

perm

Array, size (n). Depending on the value of *iparm*[4] and *iparm*[30], holds the permutation vector of size n , specifies elements used for computing a partial solution, or specifies differing values of the input matrices for low rank update.

- If *iparm*[4] = 1, *iparm*[30] = 0, and *iparm*[35] = 0, *perm* specifies the fill-in reducing ordering to the solver. Let A be the original matrix and $C = P^*A^*P^T$ be the permuted matrix. Row (column) i of C is the *perm*[i] row (column) of A . The array *perm* is also used to return the permutation vector calculated during fill-in reducing ordering stage.

NOTE

Be aware that setting *iparm*[4] = 1 prevents use of a parallel algorithm for the solve step.

- If *iparm*[4] = 2, *iparm*[30] = 0, and *iparm*[35] = 0, the permutation vector computed in phase 11 is returned in the *perm* array.
- If *iparm*[4] = 0, *iparm*[30] > 0, and *iparm*[35] = 0, *perm* specifies elements of the right-hand side to use or of the solution to compute for a partial solution.
- If *iparm*[4] = 0, *iparm*[30] = 0, and *iparm*[35] > 0, *perm* specifies elements for a Schur complement.
- If *iparm*[38] = 1, *perm* specifies values that differ in A for low rank update (see [Low Rank Update](#)). The size of the array must be at least $2*ndiff + 1$, where *ndiff* is the number of values of A that are different. The values of *perm* should be:

```
perm = {ndiff, row_index1, column_index1, row_index2,
        column_index2, ..., row_index_ndiff, column_index_ndiff}
```

where *row_index_m* and *column_index_m* are the row and column indices of the m -th differing non-zero value in matrix A . The row and column index pairs can be in any order, but must use zero-based indexing regardless of the value of *iparm*[34].

See [iparm](#)[4], [iparm](#)[30], and [iparm](#)[38] for more details.

Indexing of *perm* is one-based by default, but unless *iparm*[38] = 1 it can be changed to zero-based by setting the appropriate value to the parameter [iparm](#)[34].

nrhs

Number of right-hand sides that need to be solved for.

iparm

Array, size (64). This array is used to pass various parameters to Intel® oneAPI Math Kernel Library (oneMKL) PARDISO and to return some useful information after execution of the solver.

See [pardiso iparm Parameter](#) for more details about the *iparm* parameters.

msglvl

Message level information. If *msglvl* = 0 then *pardiso* generates no output, if *msglvl* = 1 the solver prints statistical information to the screen.

b

Array, size ($n*nrhs$). On entry, contains the right-hand side vector/matrix B , which is placed in memory contiguously. The *b*[$+k*nrhs$] element must hold the i -th component of k -th right-hand side vector. Note that *b* is only accessed in the solution phase.

Output Parameters

(See also [Intel MKL PARDISO Parameters in Tabular Form.](#))

<i>pt</i>	Handle to internal data structure.
<i>perm</i>	See the Input Parameter description of the <i>perm</i> array.
<i>iparm</i>	On output, some <i>iparm</i> values report information such as the numbers of non-zero elements in the factors. See pardiso iparm Parameter for more details about the <i>iparm</i> parameters.
<i>b</i>	On output, the array is replaced with the solution if <i>iparm</i> [5] = 1.
<i>x</i>	Array, size (<i>n*nrhs</i>). If <i>iparm</i> [5]=0 it contains solution vector/matrix <i>X</i> , which is placed contiguously in memory. The <i>x</i> [<i>i</i> + <i>k*n</i>] element must hold the <i>i</i> -th component of the <i>k</i> -th solution vector. Note that <i>x</i> is only accessed in the solution phase.
<i>error</i>	The error indicator according to the below table:

error	Information
0	no error
-1	input inconsistent
-2	not enough memory
-3	reordering problem
-4	Zero pivot, numerical factorization or iterative refinement problem. If the error appears during the solution phase, try to change the pivoting perturbation (<i>iparm</i> [9]) and also increase the number of iterative refinement steps. If it does not help, consider changing the scaling, matching and pivoting options (<i>iparm</i> [10], <i>iparm</i> [12], <i>iparm</i> [20])
-5	unclassified (internal) error
-6	reordering failed (matrix types 11 and 13 only)
-7	diagonal matrix is singular
-8	32-bit integer overflow problem
-9	not enough memory for OOC
-10	error opening OOC files
-11	read/write error with OOC files
-12	(<i>pardiso_64</i> only) <i>pardiso_64</i> called from 32-bit library
-13	interrupted by the (user-defined) mkl_progress function

error

-15

Information

internal error which can appear for `iparm[23]=10` and `iparm[12]=1`. Try switch matching off (set `iparm[12]=0` and rerun.)

pardisoinit

Initialize Intel® oneAPI Math Kernel Library (oneMKL) PARDISO with default parameters in accordance with the matrix type.

Syntax

```
void pardisoinit (_MKL_DSS_HANDLE_t pt, const MKL_INT *mtype, MKL_INT *iparm );
```

Include Files

- `mkl.h`

Description

This function initializes the solver handle `pt` for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO with zero values (as needed for the very first call of `pardiso`) and sets default `iparm` values in accordance with the matrix type `mtype`.

The recommended way is to avoid using `pardisoinit` and to initialize `pt` and set the values of the `iparm` array manually as the default parameters might not be the best for a particular use case.

An alternative method to set default `iparm` values is to call `pardiso` in the analysis phase with `iparm(1)=0`. In this case, the solver handle `pt` must be initialized with zero values.

The `pardisoinit` routine initializes only the in-core version of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. Switching to the out-of-core version of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO as well as changing default `iparm` values can be done after the call to `pardisoinit` but before the first call to `pardiso`.

The `pardisoinit` routine cannot be used together with the `pardiso_64` routine.

Input Parameters

`mtype` Matrix type. Based on this value `pardisoinit` chooses default values for the `iparm` array. Refer to the section [oneMKL PARDISO Parameters in Tabular Form](#) for more details about the default values of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

Output Parameters

`pt` Array of size 64. Handle to internal data structure. The `pardisoinit` routine nullifies the array `pt`.

NOTE

It is very important that `pt` is initialized with zero before the first call of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. After that first call you must never modify the array, because it could cause a serious memory leak or a crash.

`iparm`

Array of size 64. This array is used to set various options for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO and to return some useful information after execution of the solver. The `pardisoinit` routine fills in the `iparm` array with the default values. Refer to the section [oneMKL PARDISO Parameters in Tabular Form](#) for more details about the default values of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

pardiso_64

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides, 64-bit integer version.

Syntax

```
void pardiso_64 (_MKL_DSS_HANDLE_t pt, const long long int *maxfct, const long long int *mnum, const long long int *mtype, const long long int *phase, const long long int *n, const void *a, const long long int *ia, const long long int *ja, long long int *perm, const long long int *nrhs, long long int *iparm, const long long int *msglvl, void *b, void *x, long long int *error);
```

Include Files

- `mkh.h`

Description

`pardiso_64` is an alternative ILP64 (64-bit integer) version of the [pardiso](#) routine (see [Description](#) section for more details). The interface of `pardiso_64` is the same as the interface of `pardiso`, but it accepts and returns all integer data as `long long int`.

Use `pardiso_64` when `pardiso` for solving large matrices (with the number of non-zero elements on the order of 500 million or more). You can use it together with the usual LP64 interfaces for the rest of Intel® oneAPI Math Kernel Library (oneMKL) functionality. In other words, if you use 64-bit integer version (`pardiso_64`), you do not need to re-link your applications with ILP64 libraries. Take into account that `pardiso_64` may perform slower than regular `pardiso` on the reordering and symbolic factorization phase.

NOTE

`pardiso_64` is supported only in the 64-bit libraries. If `pardiso_64` is called from the 32-bit libraries, it returns `error == -12`.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

The input parameters of `pardiso_64` are the same as the input parameters of `pardiso`, but `pardiso_64` accepts all integer data as `long long int`.

Output Parameters

The output parameters of `pardiso_64` are the same as the [output parameters of `pardiso`](#), but `pardiso_64` returns all integer data as `long long int`.

`mkl_pardiso_pivot`

Replaces routine which handles Intel® oneAPI Math Kernel Library (oneMKL) PARDISO pivots with user-defined routine.

Syntax

```
void mkl_pardiso_pivot (const void *ai, void *bi, const void *eps);
```

Include Files

- `mkl.h`

Description

The `mkl_pardiso_pivot` routine allows you to handle diagonal elements which arise during numerical factorization that are zero or near zero. By default, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO determines that a diagonal element `bi` is a pivot if `bi < eps`, and if so, replaces it with `eps`. But you can provide your own routine to modify the resulting factorized matrix in case there are small elements on the diagonal during the factorization step.

NOTE

To use this routine, you must set `iparm[55]` to 1 before the main `pardiso` loop.

Input Parameters

<code>ai</code>	Diagonal element of initial matrix corresponding to pivot element.
<code>bi</code>	Diagonal element of factorized matrix that could be chosen as a pivot element.
<code>eps</code>	Scalar to compare with diagonal of factorized matrix. On input equal to parameter described by <code>iparm[9]</code> .

Output Parameters

<code>bi</code>	In case element is chosen as a pivot, value with which to replace the pivot.
-----------------	--

`pardiso_getdiag`

Returns diagonal elements of initial and factorized matrix.

Syntax

```
void pardiso_getdiag (const _MKL_DSS_HANDLE_t pt, void *df, void *da, const MKL_INT *mnum, MKL_INT *error);
```

Include Files

- `mkl.h`

Description

This routine returns the diagonal elements of the initial and factorized matrix for a real or Hermitian matrix.

NOTE

In order to use this routine, you must set `iparm[55]` to 1 before the main `pardiso` loop.

If `iparm[23]` is set to 10 (an improved two-level factorization algorithm for nonsymmetric matrices), Intel® oneAPI Math Kernel Library PARDISO will automatically use the classic algorithm for factorization.

Input Parameters

<code>pt</code>	Array with a size of 64. Handle to internal data structure for the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver. The entries must be set to zero prior to the first call <code>topardiso</code> . Unique for factorization.
<code>mnum</code>	Indicates the actual matrix for the solution phase of the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver. With this scalar you can define the diagonal elements of the factorized matrix that you want to obtain. The value must be: $1 \leq mnum \leq maxfct$. In most applications this value is 1.

Output Parameters

<code>df</code>	Array with a dimension of n . Contains diagonal elements of the factorized matrix after factorization.
-----------------	--

NOTE

Elements of `df` correspond to diagonal elements of matrix L computed during phase 22. Because during phase 22 Intel® oneAPI Math Kernel Library (oneMKL) PARDISO makes additional permutations to improve stability, it is possible that array `df` is not in line with the `perm` array computed during phase 11.

<code>da</code>	Array with a dimension of n . Contains diagonal elements of the initial matrix.
-----------------	---

NOTE

Elements of `da` correspond to diagonal elements of matrix L computed during phase 22. Because during phase 22 Intel® oneAPI Math Kernel Library (oneMKL) PARDISO makes additional permutations to improve stability, it is possible that array `da` is not in line with the `perm` array computed during phase 11.

<code>error</code>	The error indicator.
--------------------	----------------------

error	Information
0	no error
-1	Diagonal information not turned on before pardiso main loop (<i>iparm</i> [55]=0).

pardiso_export

Places pointers dedicated for sparse representation of a requested matrix (values, rows, and columns) into MKL PARDISO

Syntax

```
void pardiso_export (const _MKL_DSS_HANDLE_t pt, void* values, MKL_INT* rows, MKL_INT* columns, MKL_INT* step, MKL_INT* iparm, MKL_INT* error);
```

Include Files

- `mkl.h`

Description

This auxiliary routine places pointers dedicated for sparse representation of a requested matrix (values, rows, and columns) into MKL PARDISO. The matrix will be stored in the three-array variant of the compressed sparse row (CSR3 format) with 0-based indexing.

NOTE

Currently, this routine can be used only for a sparse Schur complement matrix. All parameters related to the Schur complement matrix (perm, iparm) must be set before the reordering stage of MKL PARDISO (phase = 11) is called.

Input Parameters

<i>pt</i>	Array with a size of 64. Handle to internal data structure for the Intel® MKL PARDISO solver. The entries must be set to zero prior to the first call to <code>pardiso</code> . Unique for factorization.
<i>iparm</i>	This array is used to pass various parameters to Intel® MKL PARDISO and to return some useful information after execution of the solver.
<i>step</i>	Stage indicator. These are the currently supported values:

Step value	Notes
1	Used to place pointers related to a Schur complement matrix in MKL PARDISO. The routine with <i>step</i> equal to 1 must be called between the reordering and factorization phases of MKL PARDISO.
-1	Used to clean the internal handle.

Input/Output Parameters

<i>values</i>	<p>Parameter type: input/output parameter.</p> <p>This array contains the non-zero elements of the requested matrix.</p>
<i>rows</i>	<p>Parameter type: input/output parameter.</p> <p>Array of size <code>(size + 1)</code></p> <p>For CSR3 format, <code>rows[i]</code> (<code>i < size</code>) points to the first column index of row <code>i</code> in the array columns; that is, <code>rows[i]</code> gives the index of the element in the array values that contains the first non-zero element from row <code>i</code> of the sparse matrix. The last element, <code>rows[size]</code>, is equal to the number of non-zero elements in the sparse matrix.</p>
<i>columns</i>	<p>Parameter type: input/output parameter.</p> <p>This array contains the column indices for the non-zero elements of the requested matrix.</p>
<i>error</i>	<p>Parameter type: output parameter.</p> <p>The error status:</p> <ul style="list-style-type: none"> • 0 indicates no error. • 1 indicates inconsistent input data.

Usage Example

The following C-style example demonstrates how to use the `pardiso_export` routine to get the sparse representation (that is, three-array CSR format) of a Schur complement matrix.

```
#include "mkl.h"

/*
 * Call the reordering phase of MKL PARDISO with iparm[35] set to -1 in
 * order to compute the Schur complement matrix only, or -2 to compute all
 * factorization arrays. perm array indices related to the Schur complement
 * matrix must be set to 1.
 */
phase = 11;
for ( i = 0; i < schur_size; i++ ) { perm[i] = 1.; }
iparm[35] = -1;
pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, perm, &nrhs,
        iparm, &msglvl, b, x, &error);

/*
 * After the reordering phase, iparm[35] will contain the number of non-zero
 * elements for the Schur complement matrix. Arrays dedicated to the sparse
 * representation of the Schur complement matrix must be allocated before
 * the factorization stage of MKL PARDISO is called.
 */
schur_nnz      = iparm[35];
schur_rows    = (MKL_INT *) mkl_malloc(schur_size+1, ALIGNMENT);
schur_columns = (MKL_INT *) mkl_malloc(schur_nnz , ALIGNMENT);
schur_values   = (DATA_TYPE *) mkl_malloc(schur_nnz , ALIGNMENT);

/*
 * Call to the pardiso_export routine with step equal to 1 in order to put
```

```

/* pointers related to the three-array CSR format into MKL PARDISO:
 */
pardiso_export(pt, schur_values, schur_ia, schur_ja, &step, iparm, &error);

/*
 * Call the factorization phase of PARDISO with iparm[35] equal to -1 or -2
 * to compute the Schur complement matrix:
 */
phase = 22;
iparm[35] = -1;
pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, perm, &nrhs,
        iparm, &msglvl, b, x, &error);

/*
 * After the factorization stage, schur_values, schur_rows, and
 * schur_columns will contain the Schur complement matrix in CSR3 format.
 */

```

pardiso_handle_store

Store internal structures from pardiso to a file.

Syntax

```
void pardiso_handle_store (_MKL_DSS_HANDLE_t pt, const char *dirname, MKL_INT *error);
```

Include Files

- mkl.h

Description

This function stores Intel® oneAPI Math Kernel Library (oneMKL) PARDISO structures to a file, allowing you to store Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures between the stages of the `pardiso` routine. The [pardiso_handle_restore](#) routine can restore the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures from the file.

Input Parameters

<i>pt</i>	Array with a size of 64. Handle to internal data structure.
<i>dirname</i>	String containing the name of the directory to which to write the files with the content of the internal structures. Use an empty string ("") to specify the current directory. The routine creates a file named <code>handle.pds</code> in the directory.

Output Parameters

<i>pt</i>	Handle to internal data structure.
<i>error</i>	The error indicator.

error	Information
0	No error.
-2	Not enough memory.
-10	Cannot open file for writing.

error	Information
-11	Error while writing to file.
-13	Wrong file format.

pardiso_handle_restore

Restore pardiso internal structures from a file.

Syntax

```
void pardiso_handle_restore (_MKL_DSS_HANDLE_t pt, const char *dirname, MKL_INT *error);
```

Include Files

- mkl.h

Description

This function restores Intel® oneAPI Math Kernel Library (oneMKL) PARDISO structures from a file. This allows you to restore Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures stored by `pardiso_handle_store` after a phase of the `pardiso` routine and continue execution of the next phase.

Input Parameters

<i>dirname</i>	String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.
----------------	---

Output Parameters

<i>pt</i>	Array with a dimension of 64. Handle to internal data structure.
<i>error</i>	The error indicator.

error	Information
0	No error.
-2	Not enough memory.
-10	Cannot open file for reading.
-11	Error while reading from file.
-13	Wrong file format.

pardiso_handle_delete

Delete files with pardiso internal structure data.

Syntax

```
void pardiso_handle_delete (const char *dirname, MKL_INT *error);
```

Include Files

- mkl.h

Description

This function deletes files generated with `pardiso_handle_store` that contain Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures.

Input Parameters

<i>dirname</i>	String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.
----------------	---

Output Parameters

<i>error</i>	The error indicator.
error	Information
0	No error.
-10	Cannot delete files.

`pardiso_handle_store_64`

Store internal structures from `pardiso_64` to a file.

Syntax

```
void pardiso_handle_store_64 (_MKL_DSS_HANDLE_t pt, const char *dirname, MKL_INT *error);
```

Include Files

- `mkl.h`

Description

This function stores Intel® oneAPI Math Kernel Library (oneMKL) PARDISO structures to a file, allowing you to store Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures between the stages of the `pardiso_64` routine. The `pardiso_handle_restore_64` routine can restore the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures from the file.

Input Parameters

<i>pt</i>	Array with a dimension of 64. Handle to internal data structure.
<i>dirname</i>	String containing the name of the directory to which to write the files with the content of the internal structures. Use an empty string ("") to specify the current directory. The routine creates a file named <code>handle.pds</code> in the directory.

Output Parameters

<i>pt</i>	Handle to internal data structure.
<i>error</i>	The error indicator.
error	Information
0	No error.

error	Information
-2	Not enough memory.
-10	Cannot open file for writing.
-11	Error while writing to file.
-12	Not supported in 32-bit library - routine is only supported in 64-bit libraries.
-13	Wrong file format.

pardiso_handle_restore_64

Restore pardiso_64 internal structures from a file.

Syntax

```
void pardiso_handle_restore_64 (_MKL_DSS_HANDLE_t pt, const char *dirname, MKL_INT *error);
```

Include Files

- mkl.h

Description

This function restores Intel® oneAPI Math Kernel Library (oneMKL) PARDISO structures from a file. This allows you to restore Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures stored by `pardiso_handle_store_64` after a phase of the `pardiso_64` routine and continue execution of the next phase.

Input Parameters

<i>dirname</i>	String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.
----------------	---

Input Parameters

<i>pt</i>	Array with a dimension of 64. Handle to internal data structure.
<i>error</i>	The error indicator.

error	Information
0	No error.
-2	Not enough memory.
-10	Cannot open file for reading.
-11	Error while reading from file.
-13	Wrong file format.

pardiso_handle_delete_64

Syntax

Delete files with `pardiso_64` internal structure data.

```
void pardiso_handle_delete_64 (const char *dirname, MKL_INT *error);
```

Include Files

- `mkl.h`

Description

This function deletes files generated with `pardiso_handle_store_64` that contain Intel® oneAPI Math Kernel Library (oneMKL) PARDISO internal structures.

Input Parameters

dirname String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Output Parameters

error The error indicator.

error	Information
0	No error.
-10	Cannot delete files.
-12	Not supported in 32-bit library - routine is only supported in 64-bit libraries.

oneMKL PARDISO Parameters in Tabular Form

The following table lists all parameters of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO and gives their brief descriptions.

Parameter	Type	Description	Values	Comments	In/Out
<i>pt</i>	<code>void*</code>	Solver internal data address pointer	0	Must be initialized with zeros and never be modified later	in/out
<i>maxfct</i>	<code>MKL_INT*</code>	Maximal number of factors in memory	>0	Generally used value is 1	in
<i>mnum</i>	<code>MKL_INT*</code>	The number of matrix (from 1 to <i>maxfct</i>) to solve	[1: <i>maxfct</i>]	Generally used value is 1	in
<i>mtype</i>	<code>MKL_INT*</code>	Matrix type	1	Real and structurally symmetric	in
			2	Real and symmetric positive definite	

Parameter	Type	Description	Values	Comments	In/Out
<i>phase</i>	MKL_INT*	Controls the execution of the solver For <i>iparm</i> [35] > 0, phases 331, 332, and 333 perform a different decomposition. See the <i>phase</i> parameter of pardiso for details.	-2	Real and symmetric indefinite	in
			3	Complex and structurally symmetric	
			4	Complex and Hermitian positive definite	
			-4	Complex and Hermitian indefinite	
			6	Complex and symmetric matrix	
			11	Real and nonsymmetric matrix	
			13	Complex and nonsymmetric matrix	
			11	Analysis	
			12	Analysis, numerical factorization	
			13	Analysis, numerical factorization, solve	
			22	Numerical factorization	
			23	Numerical factorization, solve	
			33	Solve, iterative refinement	
			331	<i>phase</i> =33, but only forward substitution	
			332	<i>phase</i> =33, but only diagonal substitution	
			333	<i>phase</i> =33, but only backward substitution	
<i>n</i>	MKL_INT*	Number of equations in the sparse linear system $A \cdot X = B$	0	Release internal memory for L and U of the matrix number <i>mnum</i>	in
			-1	Release all internal memory for all matrices	
<i>a</i>	void*	Contains the non-zero elements of the coefficient matrix <i>A</i>	*	The size of <i>a</i> is the same as that of <i>ja</i> , and the coefficient matrix can be either real or complex. The	in

Parameter	Type	Description	Values	Comments	In/Out
<i>ia</i> [<i>n</i>]	MKL_INT*	<i>rowIndex</i> array in CSR3 format	>=0	<p>matrix must be stored in the 3-array variation of compressed sparse row (CSR3) format with increasing values of <i>ja</i> for each row</p> <p><i>ia</i>[<i>i</i>] gives the index of the element in array <i>a</i> that contains the first non-zero element from row <i>i</i> of <i>A</i>. The last element <i>ia</i>(<i>n</i>) is taken to be equal to the number of non-zero elements in <i>A</i>.</p> <p>Note: iparm[34] indicates whether row/column indexing starts from 1 or 0.</p>	in
<i>ja</i>	MKL_INT*	<i>columns</i> array in CSR3 format	>=0	<p>The column indices for each row of <i>A</i> must be sorted in increasing order. For structurally symmetric matrices zero diagonal elements must be stored in <i>a</i> and <i>ja</i>. Zero diagonal elements should be stored for symmetric matrices, although they are not required. For symmetric matrices, the solver needs only the upper triangular part of the system.</p> <p>Note: iparm[34] indicates whether row/column indexing starts from 1 or 0.</p>	in
<i>perm</i> [<i>n</i>]	MKL_INT*	Holds the permutation vector of size <i>n</i> , specifies elements used for computing a partial solution, or specifies differing values of the input matrices for low rank update	>=0	<p>You can apply your own fill-in reducing ordering (iparm[4] = 1) or return the permutation from the solver (iparm[4] = 2).</p> <p>Let $C = P^*A^*P^T$ be the permuted matrix. Row (column) <i>i</i> of <i>C</i> is the <i>perm</i>(<i>i</i>) row (column) of <i>A</i>. The numbering of the array must describe a permutation.</p>	in/out

Parameter	Type	Description	Values	Comments	In/ Out
				<p>To specify elements for a partial solution, set <code>iparm[4] = 0</code>, <code>iparm[30] > 0</code>, and <code>iparm[35] = 0</code>.</p> <p>To specify elements for a Schur complement, set <code>iparm[4] = 0</code>, <code>iparm[30] = 0</code>, and <code>iparm[35] > 0</code>.</p> <p>To specify values that differ in <i>A</i> for low rank update (see Low Rank Update), set <code>iparm[38] = 1</code>. The size of the array must be at least $2*ndiff + 1$, where <i>ndiff</i> is the number of values of <i>A</i> that are different. The values of <i>perm</i> should be:</p> <pre>perm = {ndiff, row_index1, column_index1, row_index2, column_index2, ..., row_index_ndiff, column_index_ndiff}</pre> <p>where <i>row_index_m</i> and <i>column_index_m</i> are the row and column indices of the <i>m</i>-th differing non-zero value in matrix <i>A</i>. The row and column index pairs can be in any order, but must use zero-based indexing regardless of the value of <code>iparm[34]</code>.</p> <hr/> <p>NOTE Unless you have specified low rank update, <code>iparm[34]</code> indicates whether row/column indexing starts from 1 or 0.</p> <hr/>	
<i>nrhs</i>	MKL_INT*	Number of right-hand sides that need to be solved for	>=0	<p>Generally used value is 1</p> <p>To obtain better Intel® oneAPI Math Kernel Library (oneMKL) PARDISO performance, during the numerical factorization phase you can provide the maximum number of</p>	in

Parameter	Type	Description	Values	Comments	In/Out
<i>iparm</i> [64]	MKL_INT*	This array is used to pass various parameters to Intel® oneAPI Math Kernel Library (oneMKL) PARDISO and to return some useful information after execution of the solver (see pardiso iparm Parameter for more details)	*	right-hand sides, which can be used further during the solving phase. If <i>iparm</i> [0]=0, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO fills <i>iparm</i> [1] through <i>iparm</i> [63] with default values and uses them.	in/out
<i>msglvl</i>	MKL_INT*	Message level information	0 1	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO generates no output Intel® oneAPI Math Kernel Library (oneMKL) PARDISO prints statistical information	in
<i>b</i> [<i>n*nrhs</i>]	void*	Right-hand side vectors	*	On entry, contains the right-hand side vector/matrix <i>B</i> , which is placed contiguously in memory. The <i>b</i> [<i>i+k*n</i>] element must hold the <i>i</i> -th component of <i>k</i> -th right-hand side vector. Note that <i>b</i> is only accessed in the solution phase. On output, the array is replaced with the solution if <i>iparm</i> [5]=1.	in/out
<i>x</i> [<i>n*nrhs</i>]	void*	Solution vectors	*	On output, if <i>iparm</i> [5]=0, contains solution vector/matrix <i>X</i> which is placed contiguously in memory. The <i>x</i> [<i>i+k*n</i>] element must hold the <i>i</i> -th component of <i>k</i> -th solution vector. Note that <i>x</i> is only accessed in the solution phase.	out
<i>error</i>	MKL_INT*	Error indicator	0 -1 -2	No error Input inconsistent Not enough memory	out

Parameter	Type	Description	Values	Comments	In/ Out
			-3	Reordering problem	
			-4	Zero pivot, numerical factorization or iterative refinement problem	
			-5	Unclassified (internal) error	
			-6	Reordering failed (matrix types 11 and 13 only)	
			-7	Diagonal matrix is singular	
			-8	32-bit integer overflow problem	
			-9	Not enough memory for OOC	
			-10	Problems with opening OOC temporary files	
			-11	Read/write problems with the OOC data file	

¹⁾ See description of `PARDISO_DATA_TYPE` in [PARDISO_DATA_TYPE](#).

pardiso iparm Parameter

This table describes all individual components of the Intel® oneAPI Math Kernel Library (oneMKL) `PARDISO iparm` parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

Component	Description						
<code>iparm[0]</code>	Use default values.						
input	<table> <tr> <td>0</td><td><code>iparm[1] - iparm[63]</code> are filled with default values.</td></tr> <tr> <td>≠0</td><td>You must supply all values in components <code>iparm[1] - iparm[63]</code>.</td></tr> </table>	0	<code>iparm[1] - iparm[63]</code> are filled with default values.	≠0	You must supply all values in components <code>iparm[1] - iparm[63]</code> .		
0	<code>iparm[1] - iparm[63]</code> are filled with default values.						
≠0	You must supply all values in components <code>iparm[1] - iparm[63]</code> .						
<code>iparm[1]</code>	Fill-in reducing ordering for the input matrix.						
input	<div> Caution You can control the parallel execution of the solver by explicitly setting the <code>MKL_NUM_THREADS</code> environment variable. If fewer OpenMP threads are available than specified, the execution may slow down instead of speeding up. If <code>MKL_NUM_THREADS</code> is not defined, then the solver uses all available processors. </div> <table> <tr> <td>0</td><td>The minimum degree algorithm [Li99].</td></tr> <tr> <td>2*</td><td>The nested dissection algorithm from the METIS package [Karypis98].</td></tr> <tr> <td>3</td><td>The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Intel® oneAPI Math Kernel Library (oneMKL) PARDISO Phase 1 takes significant time.</td></tr> </table>	0	The minimum degree algorithm [Li99] .	2*	The nested dissection algorithm from the METIS package [Karypis98] .	3	The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Intel® oneAPI Math Kernel Library (oneMKL) PARDISO Phase 1 takes significant time.
0	The minimum degree algorithm [Li99] .						
2*	The nested dissection algorithm from the METIS package [Karypis98] .						
3	The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Intel® oneAPI Math Kernel Library (oneMKL) PARDISO Phase 1 takes significant time.						

Component	Description														
<p>NOTE Setting <code>iparm[1] = 3</code> prevents the use of CNR mode (<code>iparm[33] > 0</code>) because Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses dynamic parallelism.</p>															
<code>iparm[2]</code>	Reserved. Set to zero.														
<code>iparm[3]</code> input	<p>Preconditioned CGS/CG.</p> <p>This parameter controls preconditioned CGS [Sonn89] for nonsymmetric or structurally symmetric matrices and Conjugate-Gradients for symmetric matrices. <code>iparm[3]</code> has the form <code>iparm[3] = 10 * L + K</code>.</p> <table> <tr> <td>$K=0$</td><td>The factorization is always computed as required by <code>phase</code>.</td></tr> <tr> <td>$K=1$</td><td>CGS iteration replaces the computation of LU. The preconditioner is LU that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.</td></tr> <tr> <td>$K=2$</td><td>CGS iteration for symmetric positive definite matrices replaces the computation of LL^T. The preconditioner is LL^T that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.</td></tr> </table> <p>The value L controls the stopping criterion of the Krylov Subspace iteration: $\text{eps}_{\text{CGS}} = 10^{-L}$ is used in the stopping criterion $dx_i / dx_0 < \text{eps}_{\text{CGS}}$ where $dx_i = \text{inv}(L * U) * r_i$ for $K = 1$ or $dx_i = \text{inv}(L * L^T) * r_i$ for $K = 2$ and r_i is the residue at iteration i of the preconditioned Krylov Subspace iteration.</p> <p>A maximum number of 150 iterations is fixed with the assumption that the iteration will converge before consuming half the factorization time. Intermediate convergence rates and residue excursions are checked and can terminate the iteration process. If <code>phase = 23</code>, then the factorization for a given A is automatically recomputed in cases where the Krylov Subspace iteration failed, and the corresponding direct solution is returned. Otherwise the solution from the preconditioned Krylov Subspace iteration is returned. Using <code>phase = 33</code> results in an error message (<code>error=-4</code>) if the stopping criteria for the Krylov Subspace iteration can not be reached. More information on the failure can be obtained from <code>iparm[19]</code>.</p> <p>The default is <code>iparm[3]=0</code>, and other values are only recommended for an advanced user. <code>iparm[3]</code> must be greater than or equal to zero.</p> <p>Examples:</p> <table> <tr> <th><code>iparm[3]</code></th><th>Description</th></tr> <tr> <td>31</td><td>LU-preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices</td></tr> <tr> <td>61</td><td>LU-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices</td></tr> <tr> <td>62</td><td>LL^T-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric positive definite matrices</td></tr> </table>	$K=0$	The factorization is always computed as required by <code>phase</code> .	$K=1$	CGS iteration replaces the computation of LU . The preconditioner is LU that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.	$K=2$	CGS iteration for symmetric positive definite matrices replaces the computation of LL^T . The preconditioner is LL^T that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.	<code>iparm[3]</code>	Description	31	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices	61	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices	62	LL^T -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric positive definite matrices
$K=0$	The factorization is always computed as required by <code>phase</code> .														
$K=1$	CGS iteration replaces the computation of LU . The preconditioner is LU that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.														
$K=2$	CGS iteration for symmetric positive definite matrices replaces the computation of LL^T . The preconditioner is LL^T that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.														
<code>iparm[3]</code>	Description														
31	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices														
61	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices														
62	LL^T -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric positive definite matrices														
<code>iparm[4]</code> input	User permutation.														

Component	Description
	<p>This parameter controls whether user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms. Another use of this parameter is to control obtaining the fill-in reducing permutation vector calculated during the reordering stage of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.</p> <p>This option is useful for testing reordering algorithms, adapting the code to special applications problems (for instance, to move zero diagonal elements to the end of $P^*A^*P^T$), or for using the permutation vector more than once for matrices with identical sparsity structures. For definition of the permutation, see the description of the <i>perm</i> parameter.</p>
	<p>Caution You can only set one of <i>iparm</i>[4], <i>iparm</i>[30], and <i>iparm</i>[35], so be sure that the <i>iparm</i>[30] (partial solution) and the <i>iparm</i>[35] (Schur complement) parameters are 0 if you set <i>iparm</i>[4].</p>
0*	User permutation in the <i>perm</i> array is ignored.
1	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the user supplied fill-in reducing permutation from the <i>perm</i> array. <i>iparm</i> [1] is ignored.
	<p>NOTE Setting <i>iparm</i>[4] = 1 prevents use of a parallel algorithm for the solve step.</p>
2	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO returns the permutation vector computed at phase 1 in the <i>perm</i> array.
<i>iparm</i> [5] input	Write solution on <i>x</i> .
	<p>NOTE The array <i>x</i> is always used.</p>
0*	The array <i>x</i> contains the solution; right-hand side vector <i>b</i> is kept unchanged.
1	The solver stores the solution on the right-hand side <i>b</i> .
<i>iparm</i> [6] output	<p>Number of iterative refinement steps performed.</p> <p>Reports the number of iterative refinement steps that were actually performed during the solve step.</p>
<i>iparm</i> [7] input	<p>Iterative refinement step.</p> <p>On entry to the solve and iterative refinement step, <i>iparm</i>[7] must be set to the maximum number of iterative refinement steps that the solver performs.</p>
	<p>NOTE Perturbed pivots result in iterative refinement (independent of the value of <i>iparm</i>[7]) and the number of executed iterations is reported in <i>iparm</i>[6].</p>
0*	The solver automatically performs two steps of iterative refinement when perturbed pivots are obtained during the numerical factorization.

Component	Description
	<p>>0</p> <p>Maximum number of iterative refinement steps that the solver performs. The solver performs not more than the absolute value of <code>iparm[7]</code> steps of iterative refinement. The solver might stop the process before the maximum number of steps if</p> <ul style="list-style-type: none"> a satisfactory level of accuracy of the solution in terms of backward error is achieved, or if it determines that the required accuracy cannot be reached. In this case the solver returns -4 in the <code>error</code> parameter. <p>The number of executed iterations is reported in <code>iparm[6]</code>.</p>
	<p><0</p> <p>Maximum number of iterative refinement steps with a negative sign. Unlike the case above the accumulation of the residuum uses extended precision real and complex data types.</p>
	<p>NOTE Currently, this feature is only supported for sequential and OpenMP threading.</p>
<code>iparm[8]</code> input	<p>Tolerance level for the relative residual in the iterative refinement process. If set to a non-zero value, an additional criterion is used for stopping the iterative refinement:</p> $\frac{\ r\ }{\ b\ } < 10^{-iparm[8]}$ <p>If set to zero, default checks are used to determine when to stop the iterations (see <code>iparm[7]</code> description).</p>
	<p>NOTE Currently it is only used for <code>iparm[23]=1</code> or <code>10</code> and OpenMP threading.</p>
<code>iparm[9]</code> input	<p>Pivoting perturbation.</p> <p>This parameter instructs Intel® oneAPI Math Kernel Library (oneMKL) PARDISO how to handle small pivots or zero pivots for nonsymmetric matrices (<code>mtype =11</code> or <code>mtype =13</code>) and symmetric matrices (<code>mtype =-2</code>, <code>mtype =-4</code>, or <code>mtype =6</code>). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04].</p> <p>Small pivots are perturbed with $eps = 10^{-iparm[9]}$.</p> <p>The magnitude of the potential pivot is tested against a constant threshold of</p> $alpha = eps * A2 _{inf},$ <p>where $eps = 10^{-iparm[9]}$, $A2 = P * P_{MPS} * D_r * A * D_c * P$, and $A2 _{inf}$ is the infinity norm of the scaled and permuted matrix A. Any tiny pivots encountered during elimination are set to the sign $(l_{II}) * eps * A2 _{inf}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with $eps = 10^{-iparm[9]}$.</p>
	<p>13*</p> <p>The default value for nonsymmetric matrices(<code>mtype =11</code>, <code>mtype=13</code>), $eps = 10^{-13}$.</p>
	<p>8*</p> <p>The default value for symmetric indefinite matrices (<code>mtype =-2</code>, <code>mtype=-4</code>, <code>mtype=6</code>), $eps = 10^{-8}$.</p>
<code>iparm[10]</code> input	<p>Scaling vectors.</p>

Component	Description
	<p>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less than or equal to 1. This scaling method is applied only to nonsymmetric matrices (<i>mtype</i> = 11 or <i>mtype</i> = 13). The scaling can also be used for symmetric indefinite matrices (<i>mtype</i> = -2, <i>mtype</i> = -4, or <i>mtype</i> = 6) when the symmetric weighted matchings are applied (<i>iparm</i>[12] = 1).</p> <p>Use <i>iparm</i>[10] = 1 (scaling) and <i>iparm</i>[12] = 1 (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point problems. Note that in the analysis phase (<i>phase</i>=11) you must provide the numerical values of the matrix <i>A</i> in array <i>a</i> in case of scaling and symmetric weighted matching.</p>
	<p>0* Disable scaling. Default for symmetric indefinite matrices.</p>
	<p>1* Enable scaling. Default for nonsymmetric matrices.</p> <p>Scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1. This scaling method is applied to nonsymmetric matrices (<i>mtype</i> = 11, <i>mtype</i> = 13). The scaling can also be used for symmetric indefinite matrices (<i>mtype</i> = -2, <i>mtype</i> = -4, <i>mtype</i> = 6) when the symmetric weighted matchings are applied (<i>iparm</i>[12] = 1).</p> <p>Note that in the analysis phase (<i>phase</i>=11) you must provide the numerical values of the matrix <i>A</i> in case of scaling.</p>
<i>iparm</i> [11] input	<p>Solve with transposed or conjugate transposed matrix <i>A</i>.</p> <p>NOTE For real matrices, the terms <i>transposed</i> and <i>conjugate transposed</i> are equivalent.</p>
	<p>0* Solve a linear system $AX = B$.</p>
	<p>1 Solve a conjugate transposed system $A^H X = B$ based on the factorization of the matrix <i>A</i>.</p>
	<p>2 Solve a transposed system $A^T X = B$ based on the factorization of the matrix <i>A</i>.</p>
<i>iparm</i> [12] input	<p>Improved accuracy using (non-) symmetric weighted matching.</p> <p>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods and complements the alternative of using more complete pivoting techniques during the numerical factorization.</p>
	<p>0* Disable matching. Default for symmetric indefinite matrices.</p>
	<p>1* Enable matching. Default for nonsymmetric matrices.</p> <p>Maximum weighted matching algorithm to permute large elements close to the diagonal.</p> <p>It is recommended to use <i>iparm</i>[10] = 1 (scaling) and <i>iparm</i>[12] = 1 (matching) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.</p>

Component	Description				
	Note that in the analysis phase (<i>phase</i> =11) you must provide the numerical values of the matrix <i>A</i> in case of symmetric weighted matching.				
<i>iparm</i> [13] output	Number of perturbed pivots. After factorization, contains the number of perturbed pivots for the matrix types: 1, 3, 11, 13, -2, -4 and 6.				
<i>iparm</i> [14] output	Peak memory on symbolic factorization. The total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase. This value is only computed in phase 1.				
<i>iparm</i> [15] output	Permanent memory on symbolic factorization. Permanent memory from the analysis and symbolic factorization phase in kilobytes that the solver needs in the factorization and solve phases. This value is only computed in phase 1.				
<i>iparm</i> [16] output	Size of factors/Peak memory on numerical factorization and solution. This parameter provides the size in kilobytes of the total memory consumed by in-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for internal floating point arrays. This parameter is computed in phase 1. See iparm[62] for the OOC mode. The total peak memory consumed by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO is $\max(iparm[14], iparm[15] + iparm[16])$				
<i>iparm</i> [17] input/output	Report the number of non-zero elements in the factors. <table> <tr> <td><0</td><td>Enable reporting if <i>iparm</i>[17] < 0 on entry. The default value is -1.</td></tr> <tr> <td>>=0</td><td>Disable reporting.</td></tr> </table>	<0	Enable reporting if <i>iparm</i> [17] < 0 on entry. The default value is -1.	>=0	Disable reporting.
<0	Enable reporting if <i>iparm</i> [17] < 0 on entry. The default value is -1.				
>=0	Disable reporting.				
<i>iparm</i> [18] input/output	Report number of floating point operations (in 10 ⁶ floating point operations) that are necessary to factor the matrix <i>A</i> . <table> <tr> <td><0</td><td>Enable report if <i>iparm</i>[18] < 0 on entry. This increases the reordering time.</td></tr> <tr> <td>>=0 *</td><td>Disable report.</td></tr> </table>	<0	Enable report if <i>iparm</i> [18] < 0 on entry. This increases the reordering time.	>=0 *	Disable report.
<0	Enable report if <i>iparm</i> [18] < 0 on entry. This increases the reordering time.				
>=0 *	Disable report.				
<i>iparm</i> [19] output	Report CG/CGS diagnostics. <table> <tr> <td>>0</td><td>CGS succeeded, reports the number of completed iterations.</td></tr> <tr> <td><0</td><td>CG/CGS failed (<i>error</i>=-4 after the solution phase). If <i>phase</i>= 23, then the factors <i>L</i> and <i>U</i> are recomputed for the matrix <i>A</i> and the error flag <i>error</i>=0 in case of a successful factorization. If <i>phase</i> = 33, then <i>error</i> = -4 signals failure. <i>iparm</i>[19]= - <i>it_cgs</i>*10 - <i>cgs_error</i>. Possible values of <i>cgs_error</i>: 1 - fluctuations of the residuum are too large 2 - <i>dx</i>_{max}<i>it_cgs</i>/2 is too large (slow convergence) 3 - stopping criterion is not reached at <i>max_it_cgs</i> 4 - perturbed pivots caused iterative refinement</td></tr> </table>	>0	CGS succeeded, reports the number of completed iterations.	<0	CG/CGS failed (<i>error</i> =-4 after the solution phase). If <i>phase</i> = 23, then the factors <i>L</i> and <i>U</i> are recomputed for the matrix <i>A</i> and the error flag <i>error</i> =0 in case of a successful factorization. If <i>phase</i> = 33, then <i>error</i> = -4 signals failure. <i>iparm</i> [19]= - <i>it_cgs</i> *10 - <i>cgs_error</i> . Possible values of <i>cgs_error</i> : 1 - fluctuations of the residuum are too large 2 - <i>dx</i> _{max} <i>it_cgs</i> /2 is too large (slow convergence) 3 - stopping criterion is not reached at <i>max_it_cgs</i> 4 - perturbed pivots caused iterative refinement
>0	CGS succeeded, reports the number of completed iterations.				
<0	CG/CGS failed (<i>error</i> =-4 after the solution phase). If <i>phase</i> = 23, then the factors <i>L</i> and <i>U</i> are recomputed for the matrix <i>A</i> and the error flag <i>error</i> =0 in case of a successful factorization. If <i>phase</i> = 33, then <i>error</i> = -4 signals failure. <i>iparm</i> [19]= - <i>it_cgs</i> *10 - <i>cgs_error</i> . Possible values of <i>cgs_error</i> : 1 - fluctuations of the residuum are too large 2 - <i>dx</i> _{max} <i>it_cgs</i> /2 is too large (slow convergence) 3 - stopping criterion is not reached at <i>max_it_cgs</i> 4 - perturbed pivots caused iterative refinement				

Component	Description								
	5 - factorization is too fast for this matrix. It is better to use the factorization method with <code>iparm[3] = 0</code>								
<code>iparm[20]</code> input	<p>Pivoting for symmetric indefinite matrices.</p> <p>NOTE Use <code>iparm[10] = 1</code> (scaling) and <code>iparm[12] = 1</code> (matchings) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.</p> <table> <tr> <td>0</td><td>Apply 1x1 diagonal pivoting during the factorization process.</td></tr> <tr> <td>1*</td><td>Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <code>mtype=-2</code>, <code>mtype=-4</code>, or <code>mtype=6</code>.</td></tr> <tr> <td>2</td><td>Apply 1x1 diagonal pivoting during the factorization process. Using this value is the same as using <code>iparm[20] = 0</code> except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by <code>iparm[7]</code> (0 by default).</td></tr> <tr> <td>3</td><td>Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <code>mtype=-2</code>, <code>mtype=-4</code>, or <code>mtype=6</code>. Using this value is the same as using <code>iparm[20] = 1</code> except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by <code>iparm[7]</code> (0 by default).</td></tr> </table>	0	Apply 1x1 diagonal pivoting during the factorization process.	1*	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <code>mtype=-2</code> , <code>mtype=-4</code> , or <code>mtype=6</code> .	2	Apply 1x1 diagonal pivoting during the factorization process. Using this value is the same as using <code>iparm[20] = 0</code> except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by <code>iparm[7]</code> (0 by default).	3	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <code>mtype=-2</code> , <code>mtype=-4</code> , or <code>mtype=6</code> . Using this value is the same as using <code>iparm[20] = 1</code> except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by <code>iparm[7]</code> (0 by default).
0	Apply 1x1 diagonal pivoting during the factorization process.								
1*	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <code>mtype=-2</code> , <code>mtype=-4</code> , or <code>mtype=6</code> .								
2	Apply 1x1 diagonal pivoting during the factorization process. Using this value is the same as using <code>iparm[20] = 0</code> except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by <code>iparm[7]</code> (0 by default).								
3	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <code>mtype=-2</code> , <code>mtype=-4</code> , or <code>mtype=6</code> . Using this value is the same as using <code>iparm[20] = 1</code> except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by <code>iparm[7]</code> (0 by default).								
<code>iparm[21]</code> output	<p>Inertia: number of positive eigenvalues.</p> <p>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO reports the number of positive eigenvalues for symmetric indefinite matrices.</p>								
<code>iparm[22]</code> output	<p>Inertia: number of negative eigenvalues.</p> <p>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO reports the number of negative eigenvalues for symmetric indefinite matrices.</p>								
<code>iparm[23]</code> input	<p>Parallel factorization control.</p> <table> <tr> <td>0*</td><td>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the classic algorithm for factorization.</td></tr> <tr> <td>1</td><td>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses a two-level factorization algorithm. This algorithm generally improves scalability in case of parallel factorization on many OpenMP threads (more than eight).</td></tr> <tr> <td colspan="2"> <p>NOTE Disable <code>iparm[10]</code> (scaling) and <code>iparm[12] = 1</code> (matching) when using the two-level factorization algorithm. Otherwise Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the classic factorization algorithm.</p> </td></tr> <tr> <td>10</td><td>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses an improved two-level factorization algorithm for nonsymmetric matrices.</td></tr> </table>	0*	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the classic algorithm for factorization.	1	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses a two-level factorization algorithm. This algorithm generally improves scalability in case of parallel factorization on many OpenMP threads (more than eight).	<p>NOTE Disable <code>iparm[10]</code> (scaling) and <code>iparm[12] = 1</code> (matching) when using the two-level factorization algorithm. Otherwise Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the classic factorization algorithm.</p>		10	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses an improved two-level factorization algorithm for nonsymmetric matrices.
0*	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the classic algorithm for factorization.								
1	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses a two-level factorization algorithm. This algorithm generally improves scalability in case of parallel factorization on many OpenMP threads (more than eight).								
<p>NOTE Disable <code>iparm[10]</code> (scaling) and <code>iparm[12] = 1</code> (matching) when using the two-level factorization algorithm. Otherwise Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the classic factorization algorithm.</p>									
10	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses an improved two-level factorization algorithm for nonsymmetric matrices.								
<code>iparm[24]</code> input	Parallel forward/backward solve control.								

Component	Description
	<p>0* Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the following strategy for parallelizing the solving step:</p> <p>In the case of the one right-hand side, the parallelization will be performed by partitioning the matrix.</p> <p>Otherwise, the parallelization will be over the right-hand sides.</p> <p>This feature is available only for in-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (see iparm[59]).</p>
	<p>1 Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the sequential forward and backward solve.</p>
	<p>2 Independent from the number of the right-hand sides, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the parallel algorithm based on the matrix partitioning.</p> <p>This feature is available only for in-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO (see iparm[59]).</p>
iparm[25]	Reserved. Set to zero.
iparm[26]	Matrix checker.
input	<p>0* Intel® oneAPI Math Kernel Library (oneMKL) PARDISO does not check the sparse matrix representation for errors.</p>
	<p>1 Intel® oneAPI Math Kernel Library (oneMKL) PARDISO checks integer arrays <i>ia</i> and <i>ja</i>. In particular, Intel® oneAPI Math Kernel Library (oneMKL) PARDISO checks whether column indices are sorted in increasing order within each row.</p>
iparm[27]	Single or double precision Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.
input	See iparm[7] for information on controlling the precision of the refinement steps.
	<p>Important</p> <p>The iparm[27] value is stored in the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO handle between Intel® oneAPI Math Kernel Library (oneMKL) PARDISO calls, so the precision mode can be changed only during phase 1.</p>
	<p>0* Input arrays (<i>a</i>, <i>x</i> and <i>b</i>) and all internal arrays must be presented in double precision.</p>
	<p>1 Input arrays (<i>a</i>, <i>x</i> and <i>b</i>) must be presented in single precision.</p> <p>In this case all internal computations are performed in single precision.</p>
iparm[28]	Reserved. Set to zero.
iparm[29]	Number of zero or negative pivots.
output	<p>If Intel® oneAPI Math Kernel Library (oneMKL) PARDISO detects zero or negative pivot for <i>mtype</i>=2 or <i>mtype</i>=4 matrix types, the factorization is stopped. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO returns immediately with an <i>error</i> = -4, and iparm[29] reports the number of the equation where the zero or negative pivot is detected.</p> <p>Note: The returned value can be different for the parallel and sequential version in case of several zero/negative pivots.</p>
iparm[30]	Partial solve and computing selected components of the solution vectors.

Component	Description
input	<p>This parameter controls the solve step of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. It can be used if only a few components of the solution vectors are needed or if you want to reduce the computation cost at the solve step by utilizing the sparsity of the right-hand sides. To use this option the input permutation vector <i>perm</i> must be defined so that when $perm(i) = 1$ it means that either the <i>i</i>-th component in the right-hand sides is nonzero, or the <i>i</i>-th component in the solution vectors is computed, or both, depending on the value of <i>iparm</i>[30].</p> <p>The permutation vector <i>perm</i> must be present in all phases of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO software. At the reordering step, the software overwrites the input vector <i>perm</i> by a permutation vector used by the software at the factorization and solver step. If <i>m</i> is the number of components such that $perm(i) = 1$, then the last <i>m</i> components of the output vector <i>perm</i> are a set of the indices <i>i</i> satisfying the condition $perm(i) = 1$ on input.</p> <p>NOTE Turning on this option often increases the time used by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for factorization and reordering steps, but it can reduce the time required for the solver step.</p> <p>Important You can use this feature for both in-core and out-of-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO as long as <i>iparm</i>[23]=1. Otherwise, you cannot use partial solve for out-of-core mode and you will need to set <i>iparm</i>[59]=0 for in-core mode. Set the parameters <i>iparm</i>[7] (iterative refinement steps), <i>iparm</i>[3] (preconditioned CGS), <i>iparm</i>[4] (user permutation), and <i>iparm</i>[35] (Schur complement) to 0 as well.</p>
0*	Disables this option.
1	<p>it is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector <i>perm</i> is defined so that $perm(i) = 1$ means that the (<i>i</i>)-th component in the right-hand sides is nonzero. In this case Intel® oneAPI Math Kernel Library (oneMKL) PARDISO only uses the non-zero components of the right-hand side vectors and computes only corresponding components in the solution vectors. That means the <i>i</i>-th component in the solution vectors is only computed if $perm(i) = 1$.</p>
2	<p>It is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector <i>perm</i> is defined so that $perm(i) = 1$ means that the <i>i</i>-th component in the right-hand sides is nonzero.</p> <p>Unlike for <i>iparm</i>[30]=1, all components of the solution vector are computed for this setting and all components of the right-hand sides are used. Because all components are used, for <i>iparm</i>[30]=2 you must set the <i>i</i>-th component of the right-hand sides to zero explicitly if $perm(i)$ is not equal to 1.</p>
3	<p>Selected components of the solution vectors are computed. The <i>perm</i> array is not related to the right-hand sides and it only indicates which components of the solution vectors should be computed. In this case $perm(i) = 1$ means that the <i>i</i>-th component in the solution vectors is computed.</p>
<i>iparm</i> [31] - <i>iparm</i> [32]	Reserved. Set to zero.
<i>iparm</i> [33] input	Optimal number of OpenMP threads for conditional numerical reproducibility (CNR) mode.

Component	Description
	<p>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO reads the value of <code>iparm[33]</code> during the analysis phase (phase 1), so you cannot change it later.</p> <p>Because Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses C random number generator facilities during the analysis phase (phase 1) you must take these precautions to get numerically reproducible results:</p> <ul style="list-style-type: none"> Do not alter the states of the random number generators. Do not run multiple instances of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO in parallel in the analysis phase (phase 1). <p>NOTE CNR is only available for the in-core version of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO and the non-parallel version of the nested dissection algorithm. You must also:</p> <ul style="list-style-type: none"> set <code>iparm[59]</code> to 0 in order to use the in-core version, not set <code>iparm[1]</code> to 3 in order to not use the parallel version of the nested dissection algorithm. <p>Otherwise Intel® oneAPI Math Kernel Library (oneMKL) PARDISO does not produce numerically repeatable results even if CNR is enabled for Intel® oneAPI Math Kernel Library (oneMKL) using the functionality described in Support Functions for CNR.</p>
0*	CNR mode for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO is enabled only if it is enabled for Intel® oneAPI Math Kernel Library (oneMKL) using the functionality described in Support Functions for CNR and the in-core version is used. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO determines the optimal number of OpenMP threads automatically, and produces numerically reproducible results regardless of the number of threads.
>0	CNR mode is enabled for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO if in-core version is used and the optimal number of OpenMP threads for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO to rely on is defined by the value of <code>iparm[33]</code> . You can use <code>iparm[33]</code> to enable CNR mode independent from other Intel® oneAPI Math Kernel Library (oneMKL) domains. To get the best performance, set <code>iparm[33]</code> to the actual number of hardware threads dedicated for Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. Setting <code>iparm[33]</code> to fewer OpenMP threads than the maximum number of them in use reduces the scalability of the problem being solved. Setting <code>iparm[33]</code> to more threads than are available can reduce the performance of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.
<code>iparm[34]</code> input	<p>One- or zero-based indexing of columns and rows.</p> <p>NOTE Schur complement may be inaccurate or incorrect if pivots are detected. Please, check the output of <code>iparm[28]</code>.</p>
0*	One-based indexing: columns and rows indexing in arrays <code>ia</code> , <code>ja</code> , and <code>perm</code> starts from 1 (Fortran-style indexing).
1	Zero-based indexing: columns and rows indexing in arrays <code>ia</code> , <code>ja</code> , and <code>perm</code> starts from 0 (C-style indexing).

Component	Description
<i>iparm</i> [35] input/output	<p>Schur complement matrix computation control. To calculate this matrix, you must set the input permutation vector <i>perm</i> to a set of indexes such that when $perm(i) = 1$, the <i>i</i>-th element of the initial matrix is an element of the Schur matrix.</p> <hr/> <p>Caution You can only set one of <i>iparm</i>[4], <i>iparm</i>[30], and <i>iparm</i>[35], so be sure that the <i>iparm</i>[4] (user permutation) and the <i>iparm</i>[30] (partial solution) parameters are 0 if you set <i>iparm</i>[35].</p> <hr/> <p>0* Do not compute Schur complement.</p> <hr/> <p>1 Compute Schur complement matrix as part of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO factorization step and return it in the solution vector.</p> <hr/> <p>NOTE This option only computes the Schur complement matrix, and does not calculate factorization arrays.</p> <hr/> <p>2 Compute Schur complement matrix as part of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO factorization step and return it in the solution vector. Since this option calculates factorization arrays you can use it to launch partial or full solution of the entire problem after the factorization step.</p> <hr/> <p>-1 Same as <i>iparm</i>[35] equals 1, but the Schur complement matrix is provided in 3-array CSR sparse format. Use in combination with <i>pardiso_export</i>. After reordering stage of MKL PARDISO, <i>iparm</i>[35] contains number of nonzero elements for Schur complement matrix. Set it once again before calling the factorization phase.</p> <hr/> <p>NOTE This option is available only when <i>iparm</i>[23] is not equal to 0.</p> <hr/> <p>-2 Same as <i>iparm</i>[35] equals 2, but the Schur complement matrix is provided in 3-array CSR sparse format. Use in combination with <i>pardiso_export</i>. After reordering stage of MKL PARDISO, <i>iparm</i>[35] contains number of nonzero elements for Schur complement matrix. Set it once again before calling the factorization phase.</p> <hr/> <p>NOTE This option is available only when <i>iparm</i>[23] is not equal to 0.</p> <hr/>
<i>iparm</i> [36] input	<p>Format for matrix storage.</p> <hr/> <p>0* Use CSR format (see Three Array Variation of CSR Format) for matrix storage.</p> <hr/> <p>> 0 Use BSR format (see Three Array Variation of BSR Format) for matrix storage with blocks of size <i>iparm</i>[36].</p> <hr/>

Component	Description				
	<p>NOTE Intel® oneAPI Math Kernel Library (oneMKL) does not support BSR format in these cases:</p> <ul style="list-style-type: none"> • <i>iparm</i>[10] > 0: Scaling vectors • <i>iparm</i>[12] > 0: Weighted matching • <i>iparm</i>[30] > 0: Partial solution • <i>iparm</i>[35] > 0: Schur complement • <i>iparm</i>[55] > 0: Pivoting control • <i>iparm</i>[59] > 0: OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO 				
< 0	<p>Convert supplied matrix to variable BSR (VBSR) format (see Sparse Data Storage) for matrix storage. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO analyzes the matrix provided in CSR3 format and converts it to an internal VBSR format. Set <i>iparm</i>[36] = -<i>t</i>, 0 < <i>t</i> ≤ 100.</p> <p>NOTE Intel® oneAPI Math Kernel Library (oneMKL) supports only the VBSR format for real and symmetric positive definite or indefinite matrices (<i>mtype</i> = 2 or <i>mtype</i> = -2). Intel® oneAPI Math Kernel Library (oneMKL) does not support VBSR format in these cases:</p> <ul style="list-style-type: none"> • <i>iparm</i>[10] > 0: Scaling vectors • <i>iparm</i>[12] > 0: Weighted matching • <i>iparm</i>[55] > 0: Pivoting control <p>NOTE Intel® oneAPI Math Kernel Library (oneMKL) supports these features for all matrix types as long as <i>iparm</i>[23] = 1:</p> <ul style="list-style-type: none"> • <i>iparm</i>[30] > 0: Partial solution • <i>iparm</i>[35] > 0: Schur complement • <i>iparm</i>[59] > 0: OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO 				
<i>iparm</i> [37]	Reserved. Set to zero.				
<i>iparm</i> [38]	<p>Enable low rank update (see Low Rank Update) to accelerate factorization for multiple matrices with identical structure and similar values.</p> <table> <tr> <td>0*</td><td>Do not use low rank update functionality.</td></tr> <tr> <td>1</td><td> <p>Use low rank update functionality. You must also set <i>iparm</i>[23] = 10 and provide a list of changed values in the <i>perm</i> array.</p> <p>This option requires the default settings of <i>iparm</i>[3], <i>iparm</i>[4], <i>iparm</i>[5], <i>iparm</i>[27], <i>iparm</i>[30], <i>iparm</i>[35], <i>iparm</i>[36], <i>iparm</i>[55], and <i>iparm</i>[59] as well.</p> </td></tr> </table>	0*	Do not use low rank update functionality.	1	<p>Use low rank update functionality. You must also set <i>iparm</i>[23] = 10 and provide a list of changed values in the <i>perm</i> array.</p> <p>This option requires the default settings of <i>iparm</i>[3], <i>iparm</i>[4], <i>iparm</i>[5], <i>iparm</i>[27], <i>iparm</i>[30], <i>iparm</i>[35], <i>iparm</i>[36], <i>iparm</i>[55], and <i>iparm</i>[59] as well.</p>
0*	Do not use low rank update functionality.				
1	<p>Use low rank update functionality. You must also set <i>iparm</i>[23] = 10 and provide a list of changed values in the <i>perm</i> array.</p> <p>This option requires the default settings of <i>iparm</i>[3], <i>iparm</i>[4], <i>iparm</i>[5], <i>iparm</i>[27], <i>iparm</i>[30], <i>iparm</i>[35], <i>iparm</i>[36], <i>iparm</i>[55], and <i>iparm</i>[59] as well.</p>				
<i>iparm</i> [39] - <i>iparm</i> [41]	Reserved. Set to zero.				
<i>iparm</i> [42]	<p>Control parameter for the computation of the diagonal of inverse matrix.</p> <table> <tr> <td>0*</td><td>Do not compute the diagonal of inverse matrix.</td></tr> </table>	0*	Do not compute the diagonal of inverse matrix.		
0*	Do not compute the diagonal of inverse matrix.				

Component	Description
	<p>1 Intel® oneAPI Math Kernel Library (oneMKL) PARDISO computes the diagonal of the inverse matrix during the factorization phase. This feature is only available with two-level factorization algorithm (<i>iparm</i>[23] = 1) and real symmetric matrices (<i>mtype</i> = 2 or <i>mtype</i> = -2). The diagonal is returned in the solution vector.</p>
<i>iparm</i> [43] - <i>iparm</i> [54]	Reserved. Set to zero.
<i>iparm</i> [55]	<p>Diagonal and pivoting control.</p> <p>0* Internal function used to work with pivot and calculation of diagonal arrays turned off.</p> <p>1 You can use the mkl_pardiso_pivot callback routine to control pivot elements which appear during numerical factorization. Additionally, you can obtain the elements of initial matrix and factorized matrices after the <code>pardiso</code> factorization step diagonal using the pardiso_getdiag routine. This parameter can be turned on only in the in-core version of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.</p>
<i>iparm</i> [56] - <i>iparm</i> [58]	Reserved. Set to zero.
<i>iparm</i> [59] input	<p>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO mode.</p> <p><i>iparm</i>[59] switches between in-core (IC) and out-of-core (OOC) Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. OOC can solve very large problems by holding the matrix factors in files on the disk, which requires a reduced amount of main memory compared to IC.</p> <p>Unless you are operating in sequential mode, you can switch between IC and OOC modes after the reordering phase. However, you can get better Intel® oneAPI Math Kernel Library (oneMKL) PARDISO performance by setting <i>iparm</i>[59] before the reordering phase.</p> <p>The amount of memory used in OOC mode depends on the number of OpenMP threads.</p> <p>NOTE When <i>iparm</i>[59] > 0, use the <code>MKL_PARDISO_OOC_FILE_NAME</code> environment variable to store factors.</p> <p>Warning Do not increase the number of OpenMP threads used for <code>cluster_sparse_solver</code> between the first call and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.</p>
	<p>0* IC mode.</p> <p>1 IC mode is used if the total amount of RAM (in megabytes) needed for storing the matrix factors is less than sum of two values of the environment variables: <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code> (default value 2000 MB) and <code>MKL_PARDISO_OOC_MAX_SWAP_SIZE</code> (default value 0 MB); otherwise OOC mode is used. In this case amount of RAM used by OOC mode cannot exceed the value of <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code>.</p>

Component	Description
	<p>If the total peak memory needed for storing the local arrays is more than MKL_PARDISO_OOC_MAX_CORE_SIZE, increase MKL_PARDISO_OOC_MAX_CORE_SIZE if possible.</p> <hr/> <p>NOTE Conditional numerical reproducibility (CNR) is not supported for this mode.</p> <hr/>
2	<p>OOO mode.</p> <p>The OOO mode can solve very large problems by holding the matrix factors in files on the disk. Hence the amount of RAM required by OOO mode is significantly reduced compared to IC mode.</p> <p>If the total peak memory needed for storing the local arrays is more than MKL_PARDISO_OOC_MAX_CORE_SIZE, increase MKL_PARDISO_OOC_MAX_CORE_SIZE if possible.</p> <p>To obtain better Intel® oneAPI Math Kernel Library (oneMKL) PARDISO performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase.</p> <p>Refer to How to use Intel® MKL OOO PARDISO and Storage of Matrices for more details about OOO.</p>
<i>iparm</i> [60] - <i>iparm</i> [61]	Reserved. Set to zero.
<i>iparm</i> [62] output	<p>Size of the minimum OOO memory for numerical factorization and solution.</p> <p>This parameter provides the size in kilobytes of the minimum memory required by OOO Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for internal floating point arrays. This parameter is computed in phase 1.</p> <p>Total peak memory consumption of OOO Intel® oneAPI Math Kernel Library (oneMKL) PARDISO can be estimated as $\max(iparm[14], iparm[15] + iparm[62])$.</p>
<i>iparm</i> [63]	Reserved. Set to zero.

NOTE

Generally in sparse matrices, components which are equal to zero can be considered non-zero if necessary. For example, in order to make a matrix structurally symmetric, elements which are zero can be considered non-zero. See [Sparse Matrix Storage Formats](#) for an example.

Product and Performance Information

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

Notice revision #20201201

PARDISO_DATA_TYPE

The following table lists the values of PARDISO_DATA_TYPE depending on the matrix types and values of the parameter *iparm*[27].

Data type value	Matrix type <i>mtype</i>	<i>iparm</i> [27]	comments
double	1, 2, -2, 11	0	Real matrices, do
float		1	Real matrices, sin
MKL_Complex16	3, 6, 13, 4, -4	0	Complex matrices precision
MKL_Complex8		1	Complex matrices precision

Parallel Direct Sparse Solver for Clusters Interface

The Parallel Direct Sparse Solver for Clusters Interface solves large linear systems of equations with sparse matrices on clusters. It is

- high performing
- robust
- memory efficient
- easy to use

A hybrid implementation combines Message Passing Interface (MPI) technology for data exchange between parallel tasks (processes) running on different nodes, and OpenMP* technology for parallelism inside each node of the cluster. This approach effectively uses modern hardware resources such as clusters consisting of nodes with multi-core processors. The solver code is optimized for the latest Intel processors, but also performs well on clusters consisting of non-Intel processors.

Code examples are available in the Intel® oneAPI Math Kernel Library (oneMKL) installation `examples` directory.

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

Parallel Direct Sparse Solver for Clusters Interface Algorithm

Parallel Direct Sparse Solver for Clusters Interface solves a set of sparse linear equations

$$A * X = B$$

with multiple right-hand sides using a distributed LU , LL^T , LDL^T or LDL^* factorization, where A is an n -by- n matrix, and X and B are n -by- $nrhs$ matrices.

The solution comprises four tasks:

- analysis and symbolic factorization;
- numerical factorization;
- forward and backward substitution including iterative refinement;
- termination to release all internal solver memory.

The solver first computes a symmetric fill-in reducing permutation P based on the nested dissection algorithm from the METIS package [Karypis98] (included with Intel® oneAPI Math Kernel Library (oneMKL)), followed by the Cholesky or other type of factorization (depending on matrix type) [Schenk00-2] of PAP^T . The solver uses either diagonal pivoting, or 1x1 and 2x2 Bunch and Kaufman pivoting for symmetric indefinite or Hermitian matrices before finding an approximation of X by forward and backward substitution and iterative refinement.

The initial matrix A is perturbed whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal blocks. One or two passes of iterative refinement may be required to correct the effect of the perturbations. This restricted notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. For a large set of matrices from different application areas, the accuracy of this method is comparable to a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Parallel Direct Sparse Solver for Clusters additionally improves the pivoting accuracy by applying symmetric weighted matching algorithms. These methods identify large entries in the coefficient matrix A that, if permuted close to the diagonal, enable the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. The methods are based on maximum weighted matching and improve the quality of the factor in a complementary way to the alternative idea of using more complete pivoting techniques.

Parallel Direct Sparse Solver for Clusters Interface Matrix Storage

The sparse data storage in the Parallel Direct Sparse Solver for Clusters Interface follows the scheme described in the [Sparse Matrix Storage Formats](#) section using the variable ja for *columns*, ia for *rowIndex*, and a for *values*. Column indices ja must be in increasing order per row.

When an input data structure is not accessed in a call, a NULL pointer or any valid address can be passed as a placeholder for that argument.

Algorithm Parallelization and Data Distribution

Intel® oneAPI Math Kernel Library (oneMKL) Parallel Direct Sparse Solver for Clusters enables parallel execution of the solution algorithm with efficient data distribution.

The master MPI process performs the symbolic factorization phase to represent matrix A as computational tree. Then matrix A is divided among all MPI processes in a one-dimensional manner. The same distribution is used for L -factor (the lower triangular matrix in Cholesky decomposition). Matrix A and all required internal data are broadcast to subordinate MPI processes. Each MPI process fills in its own parts of L -factor with initial values of the matrix A .

Parallel Direct Sparse Solver for Clusters Interface computes all independent parts of L -factor completely in parallel. When a block of the factor must be updated by other blocks, these updates are independently passed to a temporary array on each updating MPI process. It further gathers the result into an updated block using the `MPI_Reduce()` routine. The computations within an MPI process are dynamically divided among OpenMP threads using pipelining parallelism with a combination of left- and right-looking techniques similar to those of the PARDISO* software. Level 3 BLAS operations from Intel® oneAPI Math Kernel Library (oneMKL) ensure highly efficient performance of block-to-block update operations.

During forward/backward substitutions, respective Right Hand Side (RHS) parts are distributed among all MPI processes. All these processes participate in the computation of the solution. Finally, the solution is gathered on the master MPI process.

This approach demonstrates good scalability on clusters with Infiniband* technology. Another advantage of the approach is the effective distribution of L -factor among cluster nodes. This enables the solution of tasks with a much higher number of non-zero elements than it is possible with any Symmetric Multiprocessing (SMP) in-core direct solver.

The algorithm ensures that the memory required to keep internal data on each MPI process is decreased when the number of MPI processes in a run increases. However, the solver requires that matrix A and some other internal arrays completely fit into the memory of each MPI process.

To get the best performance, run one MPI process per physical node and set the number of OpenMP* threads per node equal to the number of physical cores on the node.

NOTE

Instead of calling `MPI_Init()`, initialize MPI with `MPI_Init_thread()` and set the MPI threading level to `MPI_THREAD_FUNNELED` or higher. For details, see the code examples in `<install_dir>/examples`.

cluster_sparse_solver

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

```
void cluster_sparse_solver (_MKL_DSS_HANDLE_t pt, const MKL_INT *maxfct, const MKL_INT *mnum, const MKL_INT *mtype, const MKL_INT *phase, const MKL_INT *n, const void *a, const MKL_INT *ia, const MKL_INT *ja, MKL_INT *perm, const MKL_INT *nrhs, MKL_INT *iparm, const MKL_INT *msglvl, void *b, void *x, const int *comm, MKL_INT *error);
```

Include Files

- `mkl_cluster_sparse_solver.h`

Description

The routine `cluster_sparse_solver` calculates the solution of a set of sparse linear equations

$$A \cdot X = B$$

with single or multiple right-hand sides, using a parallel *LU*, *LDL*, or *LL^T* factorization, where *A* is an *n*-by-*n* matrix, and *X* and *B* are *n*-by-*nrhs* vectors or matrices.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters**NOTE**

Most of the input parameters (except for the *pt*, *phase*, and *comm* parameters and, for the distributed format, the *a*, *ia*, and *ja* arrays) must be set on the master MPI process only, and ignored on other processes. Other MPI processes get all required data from the master MPI process using the MPI communicator, *comm*.

pt

Array of size 64.

Handle to internal data structure. The entries must be set to zero before the first call to `cluster_sparse_solver`.

Caution

After the first call to `cluster_sparse_solver` do not modify *pt*, as that could cause a serious memory leak.

maxfct

Ignored; assumed equal to 1.

<i>mnum</i>	Ignored; assumed equal to 1.																		
<i>mtype</i>	<p>Defines the matrix type, which influences the pivoting method. The Parallel Direct Sparse Solver for Clusters solver supports the following matrices:</p> <table> <tr><td>1</td><td>real and structurally symmetric</td></tr> <tr><td>2</td><td>real and symmetric positive definite</td></tr> <tr><td>-2</td><td>real and symmetric indefinite</td></tr> <tr><td>3</td><td>complex and structurally symmetric</td></tr> <tr><td>4</td><td>complex and Hermitian positive definite</td></tr> <tr><td>-4</td><td>complex and Hermitian indefinite</td></tr> <tr><td>6</td><td>complex and symmetric</td></tr> <tr><td>11</td><td>real and nonsymmetric</td></tr> <tr><td>13</td><td>complex and nonsymmetric</td></tr> </table>	1	real and structurally symmetric	2	real and symmetric positive definite	-2	real and symmetric indefinite	3	complex and structurally symmetric	4	complex and Hermitian positive definite	-4	complex and Hermitian indefinite	6	complex and symmetric	11	real and nonsymmetric	13	complex and nonsymmetric
1	real and structurally symmetric																		
2	real and symmetric positive definite																		
-2	real and symmetric indefinite																		
3	complex and structurally symmetric																		
4	complex and Hermitian positive definite																		
-4	complex and Hermitian indefinite																		
6	complex and symmetric																		
11	real and nonsymmetric																		
13	complex and nonsymmetric																		
<i>phase</i>	<p>Controls the execution of the solver. Usually it is a two- or three-digit integer. The first digit indicates the starting phase of execution and the second digit indicates the ending phase. Parallel Direct Sparse Solver for Clusters has the following phases of execution:</p> <ul style="list-style-type: none"> • Phase 1: Fill-reduction analysis and symbolic factorization • Phase 2: Numerical factorization • Phase 3: Forward and Backward solve including optional iterative refinement • Memory release (<i>phase</i>= -1) <p>If a previous call to the routine has computed information from previous phases, execution may start at any phase. The <i>phase</i> parameter can have the following values:</p> <table> <tr><th><i>phase</i></th><th>Solver Execution Steps</th></tr> <tr><td>11</td><td>Analysis</td></tr> <tr><td>12</td><td>Analysis, numerical factorization</td></tr> <tr><td>13</td><td>Analysis, numerical factorization, solve, iterative refinement</td></tr> <tr><td>22</td><td>Numerical factorization</td></tr> <tr><td>23</td><td>Numerical factorization, solve, iterative refinement</td></tr> <tr><td>33</td><td>Solve, iterative refinement</td></tr> <tr><td>-1</td><td>Release all internal memory for all matrices</td></tr> </table>	<i>phase</i>	Solver Execution Steps	11	Analysis	12	Analysis, numerical factorization	13	Analysis, numerical factorization, solve, iterative refinement	22	Numerical factorization	23	Numerical factorization, solve, iterative refinement	33	Solve, iterative refinement	-1	Release all internal memory for all matrices		
<i>phase</i>	Solver Execution Steps																		
11	Analysis																		
12	Analysis, numerical factorization																		
13	Analysis, numerical factorization, solve, iterative refinement																		
22	Numerical factorization																		
23	Numerical factorization, solve, iterative refinement																		
33	Solve, iterative refinement																		
-1	Release all internal memory for all matrices																		
<i>n</i>	<p>Number of equations in the sparse linear systems of equations $A^*X = B$. Constraint: $n > 0$.</p>																		
<i>a</i>	<p>Array. Contains the non-zero elements of the coefficient matrix A corresponding to the indices in <i>ja</i>. The coefficient matrix can be either real or complex. The matrix must be stored in the three-array variant of the</p>																		

compressed sparse row (CSR3) or in the three-array variant of the block compressed sparse row (BSR3) format, and the matrix must be stored with increasing values of ja for each row.

For CSR3 format, the size of a is the same as that of ja . Refer to the *values* array description in [Three Array Variation of CSR Format](#) for more details.

For BSR3 format the size of a is the size of ja multiplied by the square of the block size. Refer to the *values* array description in [Three Array Variation of BSR Format](#) for more details.

NOTE

For centralized input ($iparm[39]=0$), provide the a array for the master MPI process only. For distributed assembled input ($iparm[39]=1$ or $iparm[39]=2$), provide it for all MPI processes.

Important

The column indices of non-zero elements of each row of the matrix A must be stored in increasing order.

ia

For CSR3 format, $ia[i]$ ($i < n$) points to the first column index of row i in the array ja . That is, $ia[i]$ gives the index of the element in array a that contains the first non-zero element from row i of A . The last element $ia[n]$ is taken to be equal to the number of non-zero elements in A , plus one. Refer to *rowIndex* array description in [Three Array Variation of CSR Format](#) for more details.

For BSR3 format, $ia[i]$ ($i < n$) points to the first column index of row i in the array ja . That is, $ia[i]$ gives the index of the element in array a that contains the first non-zero block from row i of A . The last element $ia[n]$ is taken to be equal to the number of non-zero blocks in A , plus one. Refer to *rowIndex* array description in [Three Array Variation of BSR Format](#) for more details.

The array ia is accessed in all phases of the solution process.

Indexing of ia is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter $iparm[34]$. For zero-based indexing, the last element $ia[n]$ is assumed to be equal to the number of non-zero elements in matrix A .

NOTE

For centralized input ($iparm[39]=0$), provide the ia array at the master MPI process only. For distributed assembled input ($iparm[39]=1$ or $iparm[39]=2$), provide it at all MPI processes.

ja

For CSR3 format, array ja contains column indices of the sparse matrix A . It is important that the indices are in increasing order per row. For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in [Three Array Variation of CSR Format](#).

For BSR3 format, array *ja* contains column indices of the sparse matrix *A*. It is important that the indices are in increasing order per row. For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for *columns* array in [Three Array Variation of BSR Format](#).

The array *ja* is accessed in all phases of the solution process.

Indexing of *ja* is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter *iparm*(35).

NOTE

For centralized input (*iparm*(40)=0), provide the *ja* array at the master MPI process only. For distributed assembled input (*iparm*(40)=1 or *iparm*(40)=2), provide it at all MPI processes.

<i>perm</i>	Ignored.
<i>nrhs</i>	Number of right-hand sides that need to be solved for.
<i>iparm</i>	<p>Array, size 64. This array is used to pass various parameters to Parallel Direct Sparse Solver for Clusters Interface and to return some useful information after execution of the solver.</p> <p>See cluster_sparse_solver iparm Parameter for more details about the <i>iparm</i> parameters.</p>
<i>msglvl</i>	<p>Message level information. If <i>msglvl</i> = 0 then <i>cluster_sparse_solver</i> generates no output, if <i>msglvl</i> = 1 the solver prints statistical information to the screen.</p> <p>Statistics include information such as the number of non-zero elements in <i>L-factor</i> and the timing for each phase.</p> <p>Set <i>msglvl</i> = 1 if you report a problem with the solver, since the additional information provided can facilitate a solution.</p>
<i>b</i>	<p>Array, size $n \times nrhs$. On entry, contains the right-hand side vector/matrix <i>B</i>, which is placed in memory contiguously. The $b[i+k*n]$ must hold the <i>i</i>-th component of <i>k</i>-th right-hand side vector. Note that <i>b</i> is only accessed in the solution phase.</p>
<i>comm</i>	<p>MPI communicator. The solver uses the Fortran MPI communicator internally. Convert the MPI communicator to Fortran using the <code>MPI_Comm_c2f()</code> function. See the examples in the <code><install_dir>/examples</code> directory.</p>

Output Parameters

<i>pt</i>	Handle to internal data structure.
<i>perm</i>	Ignored.
<i>iparm</i>	On output, some <i>iparm</i> values report information such as the numbers of non-zero elements in the factors.

See [cluster_sparse_solver iparm Parameter](#) for more details about the *iparm* parameters.

b

On output, the array is replaced with the solution if *iparm*[5] = 1.

x

Array, size ($n \times nrhs$). If *iparm*[5]=0 it contains solution vector/matrix *X*, which is placed contiguously in memory. The *x*[*i*+*k***n*] element must hold the *i*-th component of the *k*-th solution vector. Note that *x* is only accessed in the solution phase.

error

The error indicator according to the below table:

error	Information
0	no error
-1	input inconsistent
-2	not enough memory
-3	reordering problem
-4	Zero pivot, numerical factorization or iterative refinement problem. If the error appears during the solution phase, try to change the pivoting perturbation (<i>iparm</i> [9]) and also increase the number of iterative refinement steps. If it does not help, consider changing the scaling, matching and pivoting options (<i>iparm</i> [10], <i>iparm</i> [12], <i>iparm</i> [20])
-5	unclassified (internal) error
-6	reordering failed (matrix types 11 and 13 only)
-7	diagonal matrix is singular
-8	32-bit integer overflow problem
-9	not enough memory for OOC
-10	error opening OOC files
-11	read/write error with OOC files

cluster_sparse_solver_64

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

```
void cluster_sparse_solver_64 (_MKL_DSS_HANDLE_t pt, const long long int *maxfct, const long long int *mnum, const long long int *mtype, const long long int *phase, const long long int *n, const void *a, const long long int *ia, const long long int *ja, long long int *perm, const long long int *nrhs, long long int *iparm, const long long int *msglvl, void *b, void *x, const int *comm, long long int *error);
```

Include Files

- mkl_cluster_sparse_solver.h

Description

The routine `cluster_sparse_solver_64` is an alternative ILP64 (64-bit integer) version of the `cluster_sparse_solver` routine (see the Description section for more details). The interface of `cluster_sparse_solver_64` is the same as the interface of `cluster_sparse_solver`, but it accepts and returns all integer data as `long long int`.

Use `cluster_sparse_solver_64` when `cluster_sparse_solver` for solving large matrices (with the number of non-zero elements on the order of 500 million or more). You can use it together with the usual LP64 interfaces for the rest of Intel® oneAPI Math Kernel Library (oneMKL) functionality. In other words, if you use 64-bit integer version (`cluster_sparse_solver_64`), you do not need to re-link your applications with ILP64 libraries. Take into account that `cluster_sparse_solver_64` may perform slower than regular `cluster_sparse_solver` on the reordering and symbolic factorization phase.

NOTE

`cluster_sparse_solver_64` is supported only in the 64-bit libraries. If `cluster_sparse_solver_64` is called from the 32-bit libraries, it returns `error == -12`.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

The input parameters of `cluster_sparse_solver_64` are the same as the input parameters of `cluster_sparse_solver`, but `cluster_sparse_solver_64` accepts all integer data as `long long int`.

Output Parameters

The output parameters of `cluster_sparse_solver_64` are the same as the output parameters of `cluster_sparse_solver`, but `cluster_sparse_solver_64` returns all integer data as `long long int`.

`cluster_sparse_solver_get_csr_size`

Computes the (local) number of rows and (local) number of nonzero entries for (distributed) CSR data corresponding to the provided name.

Syntax

```
void cluster_sparse_solver_get_csr_size (_MKL_DSS_HANDLE_t pt, _MKL_DSS_EXPORT_DATA
name, MKL_INT *local_nrows, MKL_INT *local_nnz, const int*comm, MKL_INT *error);
```

Include Files

- `mkl_cluster_sparse_solver.h`

Description

This routine uses the internal data created during the factorization phase of `cluster_sparse_solver` for matrix *A*. The routine then:

- Computes the local number of rows and the local number of nonzeros for CSR data that correspond to the provided *name*
- Returns the computed values in `local_nrows` and `local_nnz`

It is assumed that the CSR data defined by the *name* will be distributed in the same way as the matrix *A* (as defined by *iparm*[39]) used in `cluster_sparse_solver`.

The returned values can be used for allocating CSR arrays for factors *L* and *U*, and also for allocating arrays for permutations *P* and *Q*, or scaling matrix *D* which can then be used with

`cluster_sparse_solver_set_csr_ptrs` or `cluster_sparse_solver_set_ptr` for exporting corresponding data via `cluster_sparse_solver_export`.

NOTE

Only call this routine after the factorization phase (*phase*=22) of the `cluster_sparse_solver` has been called. Neither *pt*, nor *iparm* should be changed after the preceding call to `cluster_sparse_solver`.

Input Parameters

<i>pt</i>	<p>Array with size of 64.</p> <p>Handle to internal data structure used in the prior calls to <code>cluster_sparse_solver</code>.</p>
<hr/> <p>Caution Do not modify <i>pt</i> after the calls to <code>cluster_sparse_solver</code>.</p> <hr/>	
<i>name</i>	<p>Specifies CSR data for which the output values are computed.</p> <p>SPARSE_PTLUQT_L Factor <i>L</i> from $P^*A^*Q=L^*U$.</p> <p>SPARSE_PTLUQT_U Factor <i>U</i> from $P^*A^*Q=L^*U$.</p> <p>SPARSE_DPTLUQT_L Factor <i>L</i> from $P^* (D^{-1}A) ^*Q=L^*U$.</p> <p>SPARSE_DPTLUQT_U Factor <i>U</i> from $P^* (D^{-1}A) ^*Q=L^*U$.</p>
<i>local_nrows</i>	On entry, an array of size 1.
<i>local_nnz</i>	On entry, an array of size 1.
<i>comm</i>	<p>MPI communicator. The solver uses the Fortran MPI communicator internally. Convert the MPI communicator to Fortran using the <code>MPI_Comm_c2f()</code> function. See the examples in the <code><install_dir>/examples</code> directory.</p>

Output Parameters

<i>local_nrows</i>	On output, the local number of rows for the CSR data which correspond to the <i>name</i> .				
<i>local_nnz</i>	On output, the local number of nonzero entries for the CSR data which correspond to the <i>name</i> .				
<i>error</i>	<p>The error indicator:</p> <table> <tr> <th>error</th><th>Information</th></tr> <tr> <td>0</td><td>no error</td></tr> </table>	error	Information	0	no error
error	Information				
0	no error				

Handle to internal data structure used in the prior calls to `cluster_sparse_solver`.

Caution

Do not modify *pt* after the calls to `cluster_sparse_solver`.

<i>name</i>	<p>Specifies for which CSR data the pointers are provided.</p> <p>SPARSE_PTLUQT_L Factor <i>L</i> from $P^*A^*Q=L^*U$.</p> <p>SPARSE_PTLUQT_U Factor <i>U</i> from $P^*A^*Q=L^*U$.</p> <p>SPARSE_DPTLUQT_L Factor <i>L</i> from $P^* (D^{-1}A) ^*Q=L^*U$.</p> <p>SPARSE_DPTLUQT_U Factor <i>U</i> from $P^* (D^{-1}A) ^*Q=L^*U$.</p>
<i>rowptr</i>	<p>Array of length at least (<i>local_nrows</i>+1) where <i>local_nrows</i> is the local number of rows, which can be obtained by calling <code>cluster_sparse_solver_get_csr_size</code>. This array contains row indices, such that <code>rowptr[i] - indexing</code> is the first index of row <i>i</i> in the array's <i>vals</i> and <i>colindx</i>. Here, the value of <i>indexing</i> is 0 for zero-based indexing and 1 for one-based indexing, and must be the same as it was for the matrix <i>A</i> used in the preceding calls to <code>cluster_sparse_solver</code> (also stored in <code>iparm[34]</code>).</p> <p>Refer to <i>pointerB</i> array description in CSR Format for more details.</p>
<i>colindx</i>	<p>Array of length at least <code>rowptr[local_nrows] - rowptr[0]</code>. Indexing (zero- or one-based) must be the same as for <i>rowptr</i>. For one-based indexing, the array contains the column indices plus one for each non-zero element of the matrix which corresponds to the <i>name</i>. For zero-based indexing, the array contains the column indices for each non-zero element of the matrix.</p>
<i>vals</i>	<p>Array containing non-zero elements of the matrix which corresponds to the <i>name</i>. Its length is equal to length of the <i>colindx</i> array. Refer to values array description in CSR Format for more details.</p> <p>It will be interpreted internally as <code>float*/double*(MKL_Complex8*/MKL_Complex16*)</code> depending on <i>mtype</i> (type of the matrix <i>A</i>) and <code>iparm[27]</code> (precision) specified in the preceding call to <code>cluster_sparse_solver</code>.</p>
<i>comm</i>	<p>MPI communicator. The solver uses the Fortran MPI communicator internally. Convert the MPI communicator to Fortran using the <code>MPI_Comm_c2f()</code> function. See the examples in the <code><install_dir>/examples</code> directory.</p>

Output Parameters

<i>error</i>	The error indicator:
error	Information
0	no error

error	Information
-1	<i>pt</i> is a null pointer
-2	invalid <i>pt</i>
-3	invalid <i>name</i>
-4	unsupported <i>name</i>
-9	unsupported internal code path, consider switching off non-default <i>iparm</i> parameters for <code>cluster_sparse_solver</code>
-12	internal memory error

NOTE Refer to `cl_solver_export_c.c` for an example using this functionality.

cluster_sparse_solver_set_ptr

Internally saves a provided pointer to the data corresponding to the specified name.

Syntax

```
void cluster_sparse_solver_set_ptr (_MKL_DSS_HANDLE_t pt, _MKL_DSS_EXPORT_DATA name,
void *ptr, const int *comm, MKL_INT *error);
```

Include Files

- `mkl_cluster_sparse_solver.h`

Description

This routine internally saves the input pointer, *ptr*, of the data which correspond to the provided *name*. The saved pointer can then be used for exporting corresponding data by means of `cluster_sparse_solver_export`.

NOTE

Only call this routine after the factorization phase (*phase*=22) of the `cluster_sparse_solver` has been called. Neither *pt*, nor *iparm* should be changed after the preceding call to `cluster_sparse_solver`.

Input Parameters

pt Array with size of 64.
 Handle to internal data structure used in the prior calls to `cluster_sparse_solver`.

Caution

Do not modify *pt* after the calls to `cluster_sparse_solver`.

<i>name</i>	Specifies the data for which the pointer is provided.
	SPARSE_PTLUQT_P Permutation P from $P^*A^*Q=L^*U$.
	SPARSE_PTLUQT_Q Permutation Q from $P^*A^*Q=L^*U$.
	SPARSE_DPTLUQT_P Permutation P from $P^* (D^{-1}A)^*Q=L^*U$.
	SPARSE_DPTLUQT_Q Permutation Q from $P^* (D^{-1}A)^*Q=L^*U$.
	SPARSE_DPTLUQT_D Scaling (diagonal) D from $P^* (D^{-1}A)^*Q=L^*U$.
<i>vals</i>	<p>Array containing elements of the vector representation for the data which corresponds to the <i>name</i>. Its length should be at least <i>local_nrows</i>, where <i>local_nrows</i> is the local number of rows in a corresponding matrix (obtained from <code>cluster_sparse_solver_get_csr_size</code>, for example).</p> <p>For permutations P and Q, <i>vals</i> is interpreted as <code>MKL_INT*</code>, while for the scaling it is interpreted as <code>float*/double*(MKL_Complex8*/MKL_Complex16*)</code> depending on <i>mttype</i> (type of the matrix A) and <i>iparm</i>[27] (precision) specified in the preceding call to <code>cluster_sparse_solver</code>.</p>
<i>comm</i>	<p>MPI communicator. The solver uses the Fortran MPI communicator internally. Convert the MPI communicator to Fortran using the <code>MPI_Comm_c2f()</code> function. See the examples in the <code><install_dir>/examples</code> directory.</p>

Output Parameters

<i>error</i>	The error indicator:
error	Information
0	no error
-1	<i>pt</i> is a null pointer
-2	invalid <i>pt</i>
-3	invalid <i>name</i>
-4	unsupported <i>name</i>
-9	unsupported internal code path, consider switching off non-default <i>iparm</i> parameters for <code>cluster_sparse_solver</code>
-12	internal memory error

NOTE Refer to `cl_solver_export_c.c` for an example using this functionality.

cluster_sparse_solver_export

Computes data corresponding to the specified decomposition (defined by export operation) and fills the pointers provided by calls to

`cluster_sparse_solver_set_ptr` and/or
`cluster_sparse_solver_set_csr_ptrs`.

Syntax

```
void cluster_sparse_solver_export (_MKL_DSS_HANDLE_t pt, _MKL_DSS_EXPORT_OPERATION
operation, const int *comm, MKL_INT *error);
```

Include Files

- `mkl_cluster_sparse_solver.h`

Description

This routine computes the data for the pointers of the (distributed) data to be exported (as defined the specified *operation*). It is assumed that the exported data will be distributed in the same way as the matrix *A* (as defined by *iparm[39]*) used in `cluster_sparse_solver`.

NOTE

Only call this routine after the factorization phase (*phase=22*) of the `cluster_sparse_solver` has been called. Neither *pt*, nor *iparm* should be changed after the preceding call to `cluster_sparse_solver`.

NOTE

Only call this routine after all pointers to the data required for the specified operation have been provided by means of calling `cluster_sparse_solver_set_ptr` and/or `cluster_sparse_solver_set_csr_ptrs`.

Input Parameters

<i>pt</i>	Array with size of 64. Handle to internal data structure used in the prior calls to <code>cluster_sparse_solver</code> .
-----------	---

Caution

Do not modify *pt* after the calls to `cluster_sparse_solver`.

<i>operation</i>	Specifies a particular operation which defines what data are exported
<code>SPARSE_PTLUQT</code>	Exporting data from decomposition $P^*A^*Q=L^*U$.
<code>SPARSE_DPTLUQT</code>	Exporting data from decomposition from $P^*(D^{-1}A)^*Q=L^*U$.

NOTE

Currently, for *operation*=SPARSE_DPTLUQT a real (complex) unit vector is provided for the scaling matrix *D*. Do not turn on *scaling(iparm[10]>0)* or *matching(iparm[12]>0)* in the *iparm* during the call to *cluster_sparse_solver* for this value of *operation*.

comm

MPI communicator. The solver uses the Fortran MPI communicator internally. Convert the MPI communicator to Fortran using the *MPI_Comm_c2f()* function. See the examples in the *<install_dir>/examples* directory.

Output Parameters*error*

The error indicator:

error	Information
0	no error
-1	<i>pt</i> is a null pointer
-2	invalid <i>pt</i>
-5	invalid <i>operation</i>
-6	pointers to some of the data required for the specified <i>operation</i> were not provided prior to calling <i>cluster_sparse_solver_export</i>
-9	unsupported internal code path, consider switching off non-default <i>iparm</i> parameters for <i>cluster_sparse_solver</i>
-10	unsupported case when the matrix <i>A</i> is distributed among processes with overlap in the preceding calls to <i>cluster_sparse_solver</i>
-12	internal memory error

NOTE Refer to *cl_solver_export.c.c* for an example using this functionality.

cluster_sparse_solver iparm Parameter

The following table describes all individual components of the Parallel Direct Sparse Solver for Clusters Interface *iparm* parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

Component	Description
<i>iparm</i> [0]	Use default values.
input	0 <i>iparm</i> [1] - <i>iparm</i> (64) are filled with default values.
	!=0 You must supply all values in components <i>iparm</i> [1] - <i>iparm</i> (64).

Component	Description
<i>iparm</i> [1] input	Fill-in reducing ordering for the input matrix.
	2* The nested dissection algorithm from the METIS package [Karypis98] .
	3 The parallel version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Phase 1 takes significant time.
	10 The MPI version of the nested dissection and symbolic factorization algorithms for the matrix in distributed assembled matrix input format (<i>iparm</i> [39] > 0). The input matrix for the reordering must be distributed among different MPI processes without any intersection and all MPI ranks must have at least one row of the input matrix. Use <i>iparm</i> [40] and <i>iparm</i> [41] to set the bounds of the domain. During all of Phase 1, the entire matrix is not gathered on any one process, which can decrease computation time (especially when Phase 1 takes significant time) and decrease memory usage for each MPI process on the cluster.
<p>NOTE Distributed reordering does not work if any of <i>matching</i> (<i>iparm</i>[12]=1) / <i>scaling</i> (<i>iparm</i>[10]=1) / <i>BSR format</i> (<i>iparm</i>[36]>1) / <i>Schur complement matrix computation control</i> (<i>iparm</i>[35]>0) / <i>Partial solve</i> (<i>iparm</i>[30] > 0) is turned on, or if the distributed input matrix has overlapping distribution of rows across MPI processes.</p>	
<p>NOTE If you set <i>iparm</i>[1] = 10, <i>comm</i> = -1 (MPI communicator), and if there is one MPI process, optimization and full parallelization with the OpenMP version of the nested dissection and symbolic factorization algorithms proceeds. This can decrease computation time on multi-core computers. In this case, set <i>iparm</i>[40] = 1 and <i>iparm</i>[41] = <i>n</i> for one-based indexing, or to 0 and <i>n</i> - 1, respectively, for zero-based indexing.</p>	
<i>iparm</i> [2]	Reserved. Set to zero.
<i>iparm</i> [3]	Reserved. Set to zero.
<i>iparm</i> [4] input	<p>User permutation.</p> <p>This parameter controls whether user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms. Another use of this parameter is to control obtaining the fill-in reducing permutation vector calculated during the reordering stage of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.</p> <p>This option is useful for testing reordering algorithms, adapting the code to special applications problems (for instance, to move zero diagonal elements to the end of $P^*A^*P^T$), or for using the permutation vector more than once for matrices with identical sparsity structures. For definition of the permutation, see the description of the <i>perm</i> parameter.</p>
<p>Caution You can only set one of <i>iparm</i>[4], <i>iparm</i>[30], and <i>iparm</i>[35], so be sure that the <i>iparm</i>[30] (partial solution) and the <i>iparm</i>[35] (Schur complement) parameters are 0 if you set <i>iparm</i>[4].</p>	
0	User permutation in the <i>perm</i> array is ignored.
1	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO uses the user supplied fill-in reducing permutation from the <i>perm</i> array. <i>iparm</i> [1] is ignored.
2	Intel® oneAPI Math Kernel Library (oneMKL) PARDISO returns the permutation vector computed at phase 1 in the <i>perm</i> array.

Component	Description
<i>iparm</i> [5] input	Write solution on <i>x</i> . NOTE The array <i>x</i> is always used.
	0* The array <i>x</i> contains the solution; right-hand side vector <i>b</i> is kept unchanged.
	1 The solver stores the solution on the right-hand side <i>b</i> .
<i>iparm</i> [6] output	Number of iterative refinement steps performed. Reports the number of iterative refinement steps that were actually performed during the solve step.
<i>iparm</i> [7] input	Iterative refinement step. On entry to the solve and iterative refinement step, <i>iparm</i> [7] must be set to the maximum number of iterative refinement steps that the solver performs.
	0* The solver automatically performs two steps of iterative refinement when perturbed pivots are obtained during the numerical factorization.
	>0 Maximum number of iterative refinement steps that the solver performs. The solver performs not more than the absolute value of <i>iparm</i> [7] steps of iterative refinement. The solver might stop the process before the maximum number of steps if <ul style="list-style-type: none"> • a satisfactory level of accuracy of the solution in terms of backward error is achieved, • or if it determines that the required accuracy cannot be reached. In this case Parallel Direct Sparse Solver for Clusters Interface returns -4 in the <i>error</i> parameter. The number of executed iterations is reported in <i>iparm</i> [6].
	<0 Same as above, but the accumulation of the residuum uses extended precision real and complex data types. Perturbed pivots result in iterative refinement (independent of <i>iparm</i> [7]=0) and the number of executed iterations is reported in <i>iparm</i> [6].
<i>iparm</i> [8]	Reserved. Set to zero.
<i>iparm</i> [9] input	Pivoting perturbation. This parameter instructs Parallel Direct Sparse Solver for Clusters Interface how to handle small pivots or zero pivots for nonsymmetric matrices (<i>mtype</i> =11 or <i>mtype</i> =13) and symmetric matrices (<i>mtype</i> =-2, <i>mtype</i> =-4, or <i>mtype</i> =6). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04]. Small pivots are perturbed with $\text{eps} = 10^{-iparm[9]}$. The magnitude of the potential pivot is tested against a constant threshold of $\text{alpha} = \text{eps} * A2 _{inf}$,

Component	Description
	<p>where $\text{eps} = 10^{(-\text{iparm}[9])}$, $A2 = P^*P_{\text{MPS}}^*D_r^*A^*D_c^*P$, and $\ A2\ _{\text{inf}}$ is the infinity norm of the scaled and permuted matrix A. Any tiny pivots encountered during elimination are set to the sign $(l_{\text{II}}) * \text{eps} * \ A2\ _{\text{inf}}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with $\text{eps} = 10^{(-\text{iparm}[9])}$.</p>
	<p>13* The default value for nonsymmetric matrices ($\text{mtype} = 11, \text{mtype} = 13$), $\text{eps} = 10^{-13}$.</p>
	<p>8* The default value for symmetric indefinite matrices ($\text{mtype} = -2, \text{mtype} = -4, \text{mtype} = 6$), $\text{eps} = 10^{-8}$.</p>
<code>iparm[10]</code> input	<p>Scaling vectors.</p> <p>Parallel Direct Sparse Solver for Clusters Interface uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale.</p> <p>Use <code>iparm[10] = 1</code> (scaling) and <code>iparm[12] = 1</code> (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point problems. Note that in the analysis phase ($\text{phase} = 11$) you must provide the numerical values of the matrix A in array a in case of scaling and symmetric weighted matching.</p>
	<p>0* Disable scaling. Default for symmetric indefinite matrices.</p>
	<p>1* Enable scaling. Default for nonsymmetric matrices.</p> <p>Scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1. This scaling method is applied to nonsymmetric matrices ($\text{mtype} = 11, \text{mtype} = 13$). The scaling can also be used for symmetric indefinite matrices ($\text{mtype} = -2, \text{mtype} = -4, \text{mtype} = 6$) when the symmetric weighted matchings are applied (<code>iparm[12] = 1</code>).</p> <p>Note that in the analysis phase ($\text{phase} = 11$) you must provide the numerical values of the matrix A in case of scaling.</p>
<code>iparm[11]</code>	<p>Solve with transposed or conjugate transposed matrix A.</p>
	<p>NOTE For real matrices, the terms <i>transposed</i> and <i>conjugate transposed</i> are equivalent.</p>
	<p>0* Solve a linear system $AX = B$.</p>
	<p>1 Solve a conjugate transposed system $A^H X = B$ based on the factorization of the matrix A.</p>
	<p>2 Solve a transposed system $A^T X = B$ based on the factorization of the matrix A.</p>
<code>iparm[12]</code> input	<p>Improved accuracy using (non-) symmetric weighted matching.</p> <p>Parallel Direct Sparse Solver for Clusters Interface can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods and complements the alternative of using more complete pivoting techniques during the numerical factorization.</p>
	<p>0* Disable matching. Default for symmetric indefinite matrices.</p>
	<p>1* Enable matching. Default for nonsymmetric matrices.</p>

Component	Description				
	<p>Maximum weighted matching algorithm to permute large elements close to the diagonal.</p> <p>It is recommended to use <code>iparm[10] = 1</code> (scaling) and <code>iparm[12] = 1</code> (matching) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.</p> <p>Note that in the analysis phase (<code>phase=11</code>) you must provide the numerical values of the matrix <i>A</i> in case of symmetric weighted matching.</p>				
<code>iparm[13]</code> output	<p>Number of perturbed pivots.</p> <p>After factorization, contains the number of perturbed pivots for the matrix types: 1, 3, 11, 13, -2, -4 and 6.</p>				
<code>iparm[14]</code> output	<p>Peak memory on symbolic factorization.</p> <p>The total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase.</p> <p>This value is only computed in phase 1.</p>				
<code>iparm[15]</code> output	<p>Permanent memory on symbolic factorization.</p> <p>Permanent memory from the analysis and symbolic factorization phase in kilobytes that the solver needs in the factorization and solve phases.</p> <p>This value is only computed in phase 1.</p>				
<code>iparm[16]</code> output	<p>Size of factors/Peak memory on numerical factorization and solution.</p> <p>This parameter provides the size in kilobytes of the total memory consumed by in-core Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for internal floating point arrays. This parameter is computed in phase 1. See <code>iparm[62]</code> for the OOC mode.</p> <p>The total peak memory consumed by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO is <code>imax(iparm[14], iparm[15]+iparm[16])</code></p>				
<code>iparm[17]</code> input/output	<p>Report the number of non-zero elements in the factors.</p> <table> <tr> <td><0</td><td>Enable reporting if <code>iparm[17] < 0</code> on entry. The default value is -1.</td></tr> <tr> <td>>=0</td><td>Disable reporting.</td></tr> </table>	<0	Enable reporting if <code>iparm[17] < 0</code> on entry. The default value is -1.	>=0	Disable reporting.
<0	Enable reporting if <code>iparm[17] < 0</code> on entry. The default value is -1.				
>=0	Disable reporting.				
<code>iparm[18] - iparm[19]</code>	Reserved. Set to zero.				
<code>iparm[20]</code> input	<p>Pivoting for symmetric indefinite matrices.</p> <table> <tr> <td>0</td><td>Apply 1x1 diagonal pivoting during the factorization process.</td></tr> <tr> <td>1*</td><td>Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <code>mttype=-2</code>, <code>mttype=-4</code>, or <code>mttype=6</code>.</td></tr> </table>	0	Apply 1x1 diagonal pivoting during the factorization process.	1*	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <code>mttype=-2</code> , <code>mttype=-4</code> , or <code>mttype=6</code> .
0	Apply 1x1 diagonal pivoting during the factorization process.				
1*	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <code>mttype=-2</code> , <code>mttype=-4</code> , or <code>mttype=6</code> .				
<code>iparm[21]</code> output	<p>Inertia: number of positive eigenvalues.</p> <p>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO reports the number of positive eigenvalues for symmetric indefinite matrices.</p>				
<code>iparm[22]</code> output	<p>Inertia: number of negative eigenvalues.</p> <p>Intel® oneAPI Math Kernel Library (oneMKL) PARDISO reports the number of negative eigenvalues for symmetric indefinite matrices.</p>				

Component	Description
<i>iparm</i> [23] - <i>iparm</i> [25]	Reserved. Set to zero.
<i>iparm</i> [26]	Matrix checker.
input	0* Do not check the sparse matrix representation for errors.
	1 Check integer arrays <i>ia</i> and <i>ja</i> . In particular, check whether the column indices are sorted in increasing order within each row.
<i>iparm</i> [27]	Single or double precision Parallel Direct Sparse Solver for Clusters Interface.
input	See iparm[7] for information on controlling the precision of the refinement steps.
	0* Input arrays (<i>a</i> , <i>x</i> and <i>b</i>) and all internal arrays must be presented in double precision.
	1 Input arrays (<i>a</i> , <i>x</i> and <i>b</i>) must be presented in single precision. In this case all internal computations are performed in single precision.
<i>iparm</i> [28]	Reserved. Set to zero.
<i>iparm</i> [29]	Number of zero or negative pivots.
output	If Intel® oneAPI Math Kernel Library (oneMKL) PARDISO detects zero or negative pivot for <i>mtype</i> =2 or <i>mtype</i> =4 matrix types, the factorization is stopped. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO returns immediately with an <i>error</i> = -4, and <i>iparm</i> [29] reports the number of the equation where the zero or negative pivot is detected.
	Note: The returned value can be different for the parallel and sequential version in case of several zero/negative pivots.
<i>iparm</i> [30]	Partial solve and computing selected components of the solution vectors.
input	This parameter controls the solve step of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. It can be used if only a few components of the solution vectors are needed or if you want to reduce the computation cost at the solve step by utilizing the sparsity of the right-hand sides. To use this option the input permutation vector <i>defineperm</i> so that when <i>perm</i> (<i>i</i>) = 1 it means that either the <i>i</i> -th component in the right-hand sides is nonzero, or the <i>i</i> -th component in the solution vectors is computed, or both, depending on the value of <i>iparm</i> [30].
	The permutation vector <i>perm</i> must be present in all phases of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO software. At the reordering step, the software overwrites the input vector <i>perm</i> by a permutation vector used by the software at the factorization and solver step. If <i>m</i> is the number of components such that <i>perm</i> (<i>i</i>) = 1, then the last <i>m</i> components of the output vector <i>perm</i> are a set of the indices <i>i</i> satisfying the condition <i>perm</i> (<i>i</i>) = 1 on input.
NOTE Turning on this option often increases the time used by Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for factorization and reordering steps, but it can reduce the time required for the solver step.	
Important Set the parameters iparm[7] (iterative refinement steps), iparm[3] (preconditioned CGS), iparm[4] (user permutation), and iparm[35] (Schur complement) to 0 as well.	

Component	Description
	<p>0* Disables this option.</p>
	<p>1 it is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector <i>perm</i> is defined so that $perm(i) = 1$ means that the (<i>i</i>)-th component in the right-hand sides is nonzero. In this case Intel® oneAPI Math Kernel Library (oneMKL) PARDISO only uses the non-zero components of the right-hand side vectors and computes only corresponding components in the solution vectors. That means the <i>i</i>-th component in the solution vectors is only computed if $perm(i) = 1$.</p>
	<p>2 It is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector <i>perm</i> is defined so that $perm(i) = 1$ means that the <i>i</i>-th component in the right-hand sides is nonzero.</p> <p>Unlike for <i>iparm</i>[30]=1, all components of the solution vector are computed for this setting and all components of the right-hand sides are used. Because all components are used, for <i>iparm</i>[30]=2 you must set the <i>i</i>-th component of the right-hand sides to zero explicitly if $perm(i)$ is not equal to 1.</p>
	<p>3 Selected components of the solution vectors are computed. The <i>perm</i> array is not related to the right-hand sides and it only indicates which components of the solution vectors should be computed. In this case $perm(i) = 1$ means that the <i>i</i>-th component in the solution vectors is computed.</p>
<i>iparm</i> [31] - <i>iparm</i> [33]	Reserved. Set to zero.
<i>iparm</i> [34]	One- or zero-based indexing of columns and rows.
input	<p>0* One-based indexing: columns and rows indexing in arrays <i>ia</i>, <i>ja</i>, and <i>perm</i> starts from 1 (Fortran-style indexing).</p>
	<p>1 Zero-based indexing: columns and rows indexing in arrays <i>ia</i>, <i>ja</i>, and <i>perm</i> starts from 0 (C-style indexing).</p>
<i>iparm</i> [35]	Schur complement matrix computation control. To calculate this matrix, you must set the input permutation vector <i>perm</i> to a set of indexes such that when $perm(i) = 1$, the <i>i</i> -th element of the initial matrix is an element of the Schur matrix.
	<p>Caution</p> <p>You can only set one of <i>iparm</i>[4], <i>iparm</i>[30], and <i>iparm</i>[35], so be sure that the <i>iparm</i>[4] (user permutation) and the <i>iparm</i>[30] (partial solution) parameters are 0 if you set <i>iparm</i>[35].</p>
	<p>0* Do not compute Schur complement.</p>
	<p>1 Compute Schur complement matrix as part of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO factorization step and return it in the solution vector.</p>
	<p>NOTE</p> <p>This option only computes the Schur complement matrix, and does not calculate factorization arrays.</p>
	<p>2 Compute Schur complement matrix as part of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO factorization step and return it in the solution vector. Since this option calculates factorization arrays you can use it to launch partial or full solution of the entire problem after the factorization step.</p>

Component	Description
<i>iparm</i> [36]	Format for matrix storage.
input	0* Use CSR format (see Three Array Variation of BSR Format) for matrix storage.
	1 Use CSR format (see Three Array Variation of BSR Format) for matrix storage.
	< 0 Convert supplied matrix to variable BSR (VBSR) format (see Sparse Data Storage) for matrix storage. Intel® oneAPI Math Kernel Library (oneMKL) PARDISO analyzes the matrix provided in CSR3 format and converts it to an internal VBSR format. Set <i>iparm</i> [36] = - <i>t</i> , 0 < <i>t</i> ≤ 100.
<i>iparm</i> [37] - <i>iparm</i> [38]	Reserved. Set to zero.
<i>iparm</i> [39]	Matrix input format.
input	<p>NOTE</p> <p>Performance of the reordering step of the Parallel Direct Sparse Solver for Clusters Interface is slightly better for assembled format (CSR, <i>iparm</i>[39] = 0) than for distributed format (DCSR, <i>iparm</i>[39] > 0) for the same matrices, so if the matrix is assembled on one node do not distribute it before calling <code>cluster_sparse_solver</code>.</p>
	0* Provide the matrix in usual centralized input format: the master MPI process stores all data from matrix A, with rank=0.
	1 Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix A data. Set the bounds of the domain using <i>iparm</i> [40] and <i>iparm</i> [41]. The solution vector is placed on the master process.
	2 Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix A data. Set the bounds of the domain using <i>iparm</i> [40] and <i>iparm</i> [41]. The solution vector, A, and RHS elements are distributed between processes in same manner.
	3 Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix A data. Set the bounds of the domain using <i>iparm</i> [40] and <i>iparm</i> [41]. The A and RHS elements are distributed between processes in same manner and the solution vector is the same on each process
<i>iparm</i> [40]	Beginning of input domain.
input	The number of the matrix A row, RHS element, and, for <i>iparm</i> [39]=2, solution vector that begins the input domain belonging to this MPI process.
	Only applicable to the distributed assembled matrix input format (<i>iparm</i> [39] > 0). See Sparse Matrix Storage Formats for more details.
<i>iparm</i> [41]	End of input domain.
input	The number of the matrix A row, RHS element, and, for <i>iparm</i> [39]=2, solution vector that ends the input domain belonging to this MPI process.
	Only applicable to the distributed assembled matrix input format (<i>iparm</i> [39] > 0). See Sparse Matrix Storage Formats for more details.
<i>iparm</i> [42] - <i>iparm</i> [58]	Reserved. Set to zero.

Component	Description
input	
<i>iparm</i> [59]	cluster_sparse_solver mode.
input	<p><i>iparm</i>[59] switches between in-core (IC) and out-of-core (OOC) of cluster_sparse_solver. OOC can solve very large problems by holding the matrix factors in files on the disk, which requires a reduced amount of main memory compared to IC.</p> <p>Unless you are operating in sequential mode, you can switch between IC and OOC modes after the reordering phase. However, you can get better cluster_sparse_solver performance by setting <i>iparm</i>[59] before the reordering phase.</p> <p>The amount of memory used in OOC mode depends on the number of OpenMP threads.</p>
	<p>NOTE</p> <p>When <i>iparm</i>[59] > 0, use the MKL_PARDISO_OOC_FILE_NAME environment variable to store factors.</p>
	<p>Warning</p> <p>Do not increase the number of OpenMP threads used for cluster_sparse_solver between the first call and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.</p>
0*	IC mode.
1	<p>IC mode is used if the total amount of RAM (in megabytes) needed for storing the matrix factors is less than sum of two values of the environment variables: MKL_PARDISO_OOC_MAX_CORE_SIZE (default value 2000 MB) and MKL_PARDISO_OOC_MAX_SWAP_SIZE (default value 0 MB); otherwise OOC mode is used. In this case amount of RAM used by OOC mode cannot exceed the value of MKL_PARDISO_OOC_MAX_CORE_SIZE.</p> <p>If the total peak memory needed for storing the local arrays is more than MKL_PARDISO_OOC_MAX_CORE_SIZE, increase MKL_PARDISO_OOC_MAX_CORE_SIZE if possible.</p>
	<p>NOTE</p> <p>Conditional numerical reproducibility (CNR) is not supported for this mode.</p>
2	<p>OOC mode.</p> <p>The OOC mode can solve very large problems by holding the matrix factors in files on the disk. Hence the amount of RAM required by OOC mode is significantly reduced compared to IC mode.</p> <p>If the total peak memory needed for storing the local arrays is more than MKL_PARDISO_OOC_MAX_CORE_SIZE, increase MKL_PARDISO_OOC_MAX_CORE_SIZE if possible.</p> <p>To obtain better cluster_sparse_solver performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase.</p>

Component	Description
NOTE To use OOC mode, you must disable <code>iparm[10]</code> (scaling) and <code>iparm[12] = 1</code> (matching).	
<code>iparm[60]</code> - <code>iparm[61]</code> input	Reserved. Set to zero.
<code>iparm[62]</code> output	Size of the minimum OOC memory for numerical factorization and solution. This parameter provides the size in kilobytes of the minimum memory required by OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO for internal floating point arrays. This parameter is computed in phase 1. Total peak memory consumption of OOC Intel® oneAPI Math Kernel Library (oneMKL) PARDISO can be estimated as $\text{as}_{\max}(\text{iparm}[14], \text{iparm}[15] + \text{iparm}[62])$.
<code>iparm[63]</code> input	Reserved. Set to zero.

NOTE

Generally in sparse matrices, components which are equal to zero can be considered non-zero if necessary. For example, in order to make a matrix structurally symmetric, elements which are zero can be considered non-zero. See [Sparse Matrix Storage Formats](#) for an example.

Product and Performance Information

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

Notice revision #20201201

Direct Sparse Solver (DSS) Interface Routines

Intel® oneAPI Math Kernel Library (oneMKL) supports the DSS interface, an alternative to the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO interface for the direct sparse solver. The DSS interface implements a group of user-callable routines that are used in the step-by-step solving process and utilizes the general scheme described in [Appendix A Linear Solvers Basics](#) for solving sparse systems of linear equations. This interface also includes one routine for gathering statistics related to the solving process.

The DSS interface also supports the out-of-core (OOC) mode.

Table "DSS Interface Routines" lists the names of the routines and describes their general use.

DSS Interface Routines

Routine	Description
<code>dss_create</code>	Initializes the solver and creates the basic data structures necessary for the solver. This routine must be called before any other DSS routine.
<code>dss_define_structure</code>	Informs the solver of the locations of the non-zero elements of the matrix.

Routine	Description
<code>dss_reorder</code>	Based on the non-zero structure of the matrix, computes a permutation vector to reduce fill-in during the factoring process.
<code>dss_factor_real</code> , <code>dss_factor_complex</code>	Computes the LU , LDL^T or LL^T factorization of a real or complex matrix.
<code>dss_solve_real</code> , <code>dss_solve_complex</code>	Computes the solution vector for a system of equations based on the factorization computed in the previous phase.
<code>dss_delete</code>	Deletes all data structures created during the solving process.
<code>dss_statistics</code>	Returns statistics about various phases of the solving process.

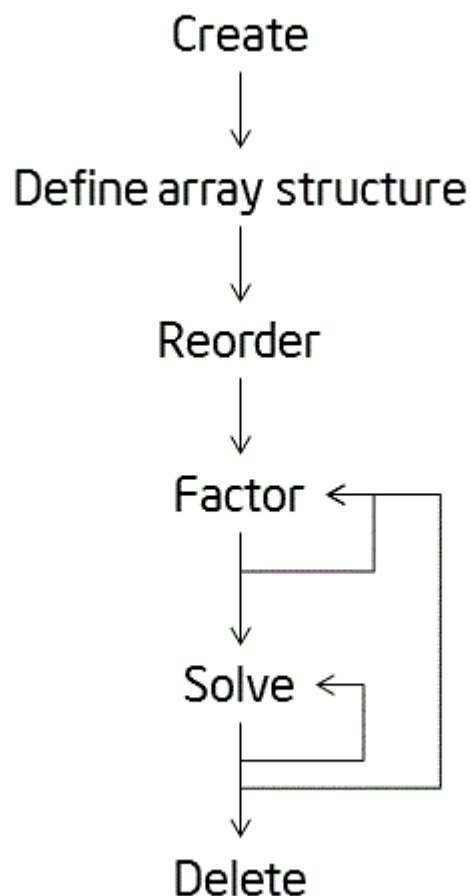
To find a single solution vector for a single system of equations with a single right-hand side, invoke the Intel® oneAPI Math Kernel Library (oneMKL) DSS interface routines in this order:

1. `dss_create`
2. `dss_define_structure`
3. `dss_reorder`
4. `dss_factor_real`, `dss_factor_complex`
5. `dss_solve_real`, `dss_solve_complex`
6. `dss_delete`

However, in certain applications it is necessary to produce solution vectors for multiple right-hand sides for a given factorization and/or factor several matrices with the same non-zero structure. Consequently, it is sometimes necessary to invoke the Intel® oneAPI Math Kernel Library (oneMKL) sparse routines in an order other than that listed, which is possible using the DSS interface. The solving process is conceptually divided into six phases. [Figure "Typical order for invoking DSS interface routines"](#) indicates the typical order in which the DSS interface routines can be invoked.

[__border__top](#)

Typical order for invoking DSS interface routines



See the code examples that use the DSS interface routines to solve systems of linear equations in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory (`dss_*.c`).

- `examples/solverc/source`

DSS Interface Description

Each DSS routine reads from or writes to a data object called a *handle*. Refer to [Memory Allocation and Handles](#) to determine the correct method for declaring a handle argument for each language. For simplicity, the descriptions in DSS routines refer to the data type as `MKL_DSS_HANDLE`.

Routine Options

The DSS routines have an integer argument (referred below to as *opt*) for passing various options to the routines. The permissible values for *opt* should be specified using only the symbol constants defined in the language-specific header files (see [Implementation Details](#)). The routines accept options for setting the message and termination levels as described in [Table "Symbolic Names for the Message and Termination Levels Options"](#). Additionally, each routine accepts the option `MKL_DSS_DEFAULTS` that sets the default values (as documented) for *opt* to the routine.

Symbolic Names for the Message and Termination Levels Options

Message Level	Termination Level
MKL_DSS_MSG_LVL_SUCCESS	MKL_DSS_TERM_LVL_SUCCESS
MKL_DSS_MSG_LVL_INFO	MKL_DSS_TERM_LVL_INFO
MKL_DSS_MSG_LVL_WARNING	MKL_DSS_TERM_LVL_WARNING
MKL_DSS_MSG_LVL_ERROR	MKL_DSS_TERM_LVL_ERROR
MKL_DSS_MSG_LVL_FATAL	MKL_DSS_TERM_LVL_FATAL

The settings for message and termination levels can be set on any call to a DSS routine. However, once set to a particular level, they remain at that level until they are changed in another call to a DSS routine.

You can specify both message and termination level for a DSS routine by adding the options together. For example, to set the message level to `debug` and the termination level to `error` for all the DSS routines, use the following call:

```
dss_create( handle, MKL_DSS_MSG_LVL_INFO + MKL_DSS_TERM_LVL_ERROR)
```

User Data Arrays

Many of the DSS routines take arrays of user data as input. For example, pointers to user arrays are passed to the routine `dss_define_structure` to describe the location of the non-zero entries in the matrix.

Caution

Do not modify the contents of these arrays after they are passed to one of the solver routines.

DSS Implementation Details

To promote portability across platforms and ease of use across different languages, use the `mkl_dss.h` header file.

The header file defines symbolic constants for returned error values, function options, certain defined data types, and function prototypes.

NOTE

Constants for options, returned error values, and message severities must be referred only by the symbolic names that are defined in these header files. Use of the Intel® oneAPI Math Kernel Library (oneMKL) DSS software without including one of the above header files is not supported.

Memory Allocation and Handles

You do not need to allocate any temporary working storage in order to use the Intel® oneAPI Math Kernel Library (oneMKL) DSS routines, because the solver itself allocates any required storage. To enable multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using *ahandle* data object.

Each of the Intel® oneAPI Math Kernel Library (oneMKL) DSS routines creates, uses, or deletes a handle. Consequently, any program calling an Intel® oneAPI Math Kernel Library (oneMKL) DSS routine must be able to allocate storage for a handle. The exact syntax for allocating storage for a handle varies from language to language. To standardize the handle declarations, the language-specific header files declare constants and defined data types that must be used when declaring a handle object in your code.

```
#include "mkl_dss.h"
_MKL_DSS_HANDLE_t handle;
```

In addition to the definition for the correct declaration of a handle, the include file also defines the following:

- function prototypes for languages that support prototypes
- symbolic constants that are used for the returned error values
- user options for the solver routines
- constants indicating message severity.

DSS Routines

dss_create

Initializes the solver.

Syntax

```
MKL_INT dss_create(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt)
```

Include Files

- mkl.h

Description

The `dss_create` routine initializes the solver. After the call to `dss_create`, all subsequent invocations of the Intel® oneAPI Math Kernel Library (oneMKL) DSS routines must use the value of the handle returned by `dss_create`.

WARNING

Do not write the value of handle directly.

The default value of the parameter `opt` is

```
MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR.
```

By default, the DSS routines use double precision for solving systems of linear equations. The precision used by the DSS routines can be set to single mode by adding the following value to the `opt` parameter:

```
MKL_DSS_SINGLE_PRECISION.
```

Input data and internal arrays are required to have single precision.

By default, the DSS routines use Fortran style (one-based) indexing for input arrays of integer types (the first value is referenced as array element 1). To set indexing to C style (the first value is referenced as array element 0), add the following value to the `opt` parameter:

```
MKL_DSS_ZERO_BASED_INDEXING.
```

The `opt` parameter can also control number of refinement steps used on the solution stage by specifying the two following values:

`MKL_DSS_REFINEMENT_OFF` - maximum number of refinement steps is set to zero;

`MKL_DSS_REFINEMENT_ON` (default value) - maximum number of refinement steps is set to 2.

By default, DSS uses in-core computations. To launch the out-of-core version of DSS (OOC DSS) you can add to this parameter one of two possible values: `MKL_DSS_OOC_STRONG` and `MKL_DSS_OOC_VARIABLE`.

`MKL_DSS_OOC_STRONG` - OOC DSS is used.

`MKL_DSS_OOC_VARIABLE` - if the memory needed for the matrix factors is less than the value of the environment variable `MKL_PARDISO_OOC_MAX_CORE_SIZE`, then the OOC DSS uses the in-core kernels of Intel® oneAPI Math Kernel Library (oneMKL) PARDISO, otherwise it uses the OOC computations.

The variable `MKL_PARDISO_OOC_MAX_CORE_SIZE` defines the maximum size of RAM allowed for storing work arrays associated with the matrix factors. It is ignored if `MKL_DSS_OOC_STRONG` is set. The default value of `MKL_PARDISO_OOC_MAX_CORE_SIZE` is 2000 MB. This value and default path and file name for storing temporary data can be changed using the configuration file `pardiso_ooc.cfg` or command line (See more details in the description of the [pardiso](#) routine).

WARNING

Other than message and termination level options, do not change the OOC DSS settings after they are specified in the routine `dss_create`.

Input Parameters

opt Parameter to pass the DSS options. The default value is `MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR`.

Output Parameters

handle Pointer to the data structure storing internal DSS results (`MKL_DSS_HANDLE`).

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_INVALID_OPTION`

`MKL_DSS_OUT_OF_MEMORY`

`MKL_DSS_MSG_LVL_ERR`

`MKL_DSS_TERM_LVL_ERR`

dss_define_structure

Communicates locations of non-zero elements in the matrix to the solver.

Syntax

```
MKL_INT dss_define_structure(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, MKL_INT
const *rowIndex, MKL_INT const *nRows, MKL_INT const *nCols, MKL_INT const *columns,
MKL_INT const *nNonZeros);
```

Include Files

- `mkh.h`

Description

The routine `dss_define_structure` communicates the locations of the *nNonZeros* number of non-zero elements in a matrix of *nRows* * *nCols* size to the solver.

NOTE

The Intel® oneAPI Math Kernel Library (oneMKL) DSS software operates only on square matrices, so *nRows* must be equal to *nCols*.

To communicate the locations of non-zero elements in the matrix, do the following:

1. Define the general non-zero structure of the matrix by specifying the value for the options argument *opt*. You can set the following values for real matrices:

- MKL_DSS_SYMMETRIC_STRUCTURE
- MKL_DSS_SYMMETRIC
- MKL_DSS_NON_SYMMETRIC

and for complex matrices:

- MKL_DSS_SYMMETRIC_STRUCTURE_COMPLEX
- MKL_DSS_SYMMETRIC_COMPLEX
- MKL_DSS_NON_SYMMETRIC_COMPLEX

The information about the matrix type must be defined in `dss_define_structure`.

2. Provide the actual locations of the non-zeros by means of the arrays *rowIndex* and *columns* (see [Sparse Matrix Storage Format](#)).

NOTE No diagonal element can be omitted from the values array. If there is a zero value on the diagonal, for example, that element nonetheless must be explicitly represented.

Input Parameters

<i>opt</i>	Parameter to pass the DSS options. The default value for the matrix structure is <code>MKL_DSS_SYMMETRIC</code> .
<i>rowIndex</i>	Array of size <i>nRows</i> +1. Defines the location of non-zero entries in the matrix.
<i>nRows</i>	Number of rows in the matrix.
<i>nCols</i>	Number of columns in the matrix; must be equal to <i>nRows</i> .
<i>columns</i>	Array of size <i>nNonZeros</i> . Defines the column location of non-zero entries in the matrix.
<i>nNonZeros</i>	Number of non-zero elements in the matrix.

Output Parameters

<i>handle</i>	Pointer to the data structure storing internal DSS results (<code>MKL_DSS_HANDLE</code>).
---------------	---

Return Values

`MKL_DSS_SUCCESS`
`MKL_DSS_STATE_ERR`
`MKL_DSS_INVALID_OPTION`
`MKL_DSS_STRUCTURE_ERR`
`MKL_DSS_ROW_ERR`
`MKL_DSS_COL_ERR`
`MKL_DSS_NOT_SQUARE`

MKL_DSS_TOO_FEW_VALUES
 MKL_DSS_TOO_MANY_VALUES
 MKL_DSS_OUT_OF_MEMORY
 MKL_DSS_MSG_LVL_ERR
 MKL_DSS_TERM_LVL_ERR

dss_reorder

Computes or sets a permutation vector that minimizes the fill-in during the factorization phase.

Syntax

```
MKL_INT dss_reorder(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, MKL_INT const *perm)
```

Include Files

- mkl.h

Description

If *opt* contains the option MKL_DSS_AUTO_ORDER, then the routine `dss_reorder` computes a permutation vector that minimizes the fill-in during the factorization phase. For this option, the routine ignores the contents of the *perm* array.

If *opt* contains the option MKL_DSS_METIS_OPENMP_ORDER, then the routine `dss_reorder` computes permutation vector using the parallel nested dissections algorithm to minimize the fill-in during the factorization phase. This option can be used to decrease the time of `dss_reorder` call on multi-core computers. For this option, the routine ignores the contents of the *perm* array.

If *opt* contains the option MKL_DSS_MY_ORDER, then you must supply a permutation vector in the array *perm*. In this case, the array *perm* is of length *nRows*, where *nRows* is the number of rows in the matrix as defined by the previous call to `dss_define_structure`.

If *opt* contains the option MKL_DSS_GET_ORDER, then the permutation vector computed during the `dss_reorder` call is copied to the array *perm*. In this case you must allocate the array *perm* beforehand. The permutation vector is computed in the same way as if the option MKL_DSS_AUTO_ORDER is set.

Product and Performance Information

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

Notice revision #20201201

Input Parameters

<i>opt</i>	Parameter to pass the DSS options. The default value for the permutation type is MKL_DSS_AUTO_ORDER.
<i>perm</i>	Array of length <i>nRows</i> . Contains a user-defined permutation vector (accessed only if <i>opt</i> contains MKL_DSS_MY_ORDER or MKL_DSS_GET_ORDER).

Output Parameters

handle Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_REORDER_ERR

MKL_DSS_REORDER1_ERR

MKL_DSS_I32BIT_ERR

MKL_DSS_FAILURE

MKL_DSS_OUT_OF_MEMORY

MKL_DSS_MSG_LVL_ERR

MKL_DSS_TERM_LVL_ERR

dss_factor_real, dss_factor_complex

Compute factorization of the matrix with previously specified location of non-zero elements.

Syntax

```
MKL_INT dss_factor_real(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, void const *rValues)
```

```
MKL_INT dss_factor_complex(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, void const *cValues)
```

Include Files

- mkl.h

Description

These routines compute factorization of the matrix whose non-zero locations were previously specified by a call to [dss_define_structure](#) and whose non-zero values are given in the array *rValues*, *cValues* or *Values*. Data type These arrays must be of length *nNonZeros* as defined in a previous call to [dss_define_structure](#).

NOTE

The data type (single or double precision) of *rValues*, *cValues*, *Values* must be in correspondence with precision specified by the parameter *opt* in the routine [dss_create](#).

The *opt* argument can contain one of the following options:

- MKL_DSS_POSITIVE_DEFINITE
- MKL_DSS_INDEFINITE
- MKL_DSS_HERMITIAN_POSITIVE_DEFINITE

- MKL_DSS_HERMITIAN_INDEFINITE

depending on your matrix's type.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) for details.

Input Parameters

<i>handle</i>	Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).
<i>opt</i>	Parameter to pass the DSS options. The default value is MKL_DSS_POSITIVE_DEFINITE.
<i>rValues</i>	Array of elements of the matrix <i>A</i> . Real data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <i>dss_create</i> .
<i>cValues</i>	Array of elements of the matrix <i>A</i> . Complex data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <i>dss_create</i> .

Return Values

MKL_DSS_SUCCESS
 MKL_DSS_STATE_ERR
 MKL_DSS_INVALID_OPTION
 MKL_DSS_OPTION_CONFLICT
 MKL_DSS_VALUES_ERR
 MKL_DSS_OUT_OF_MEMORY
 MKL_DSS_ZERO_PIVOT
 MKL_DSS_FAILURE
 MKL_DSS_MSG_LVL_ERR
 MKL_DSS_TERM_LVL_ERR
 MKL_DSS_OOC_MEM_ERR
 MKL_DSS_OOC_OC_ERR
 MKL_DSS_OOC_RW_ERR

dss_solve_real, dss_solve_complex

Compute the corresponding solution vector and place it in the output array.

Syntax

```

MKL_INT dss_solve_real(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, void const
*rRhsValues, MKL_INT const *nRhs, void *rSolValues)

MKL_INT dss_solve_complex(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, void const
*cRhsValues, MKL_INT const *nRhs, void *cSolValues)

```

Include Files

- `mk1.h`

Description

For each right-hand side column vector defined in the arrays *rRhsValues*, *cRhsValues*, or *RhsValues*, these routines compute the corresponding solution vector and place it in the arrays *rSolValues*, *cSolValues*, or *SolValues* respectively.

NOTE

The data type (single or double precision) of all arrays must be in correspondence with precision specified by the parameter *opt* in the routine `dss_create`.

The lengths of the right-hand side and solution vectors, *nRows* and *nCols* respectively, must be defined in a previous call to `dss_define_structure`.

By default, both routines perform the full solution step (it corresponds to *phase* = 33 in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO). The parameter *opt* enables you to calculate the final solution step-by-step, calling forward and backward substitutions.

If it is set to `MKL_DSS_FORWARD_SOLVE`, the forward substitution (corresponding to *phase* = 33 in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO) is performed;

if it is set to `MKL_DSS_DIAGONAL_SOLVE`, the diagonal substitution (corresponding to *phase* = 332 in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO) is performed, if possible;

if it is set to `MKL_DSS_BACKWARD_SOLVE`, the backward substitution (corresponding to *phase* = 333 in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO) is performed.

For more details about using these substitutions for different types of matrices, see [Separate Forward and Backward Substitution](#) in the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver description.

This parameter also can control the number of refinement steps that is used on the solution stage: if it is set to `MKL_DSS_REFINEMENT_OFF`, the maximum number of refinement steps equal to zero, and if it is set to `MKL_DSS_REFINEMENT_ON` (default value), the maximum number of refinement steps is equal to 2.

`MKL_DSS_CONJUGATE_SOLVE` option added to the parameter *opt* enables solving a conjugate transposed system $A^H X = B$ based on the factorization of the matrix *A*. This option is equivalent to the parameter *iparm*[11] = 1 in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

`MKL_DSS_TRANSPOSE_SOLVE` option added to the parameter *opt* enables solving a transposed system $A^T X = B$ based on the factorization of the matrix *A*. This option is equivalent to the parameter *iparm*[11] = 2 in Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

Input Parameters

<i>handle</i>	Pointer to the data structure storing internal DSS results (<code>MKL_DSS_HANDLE</code>).
<i>opt</i>	Parameter to pass the DSS options.
<i>nRhs</i>	Number of the right-hand sides in the system of linear equations.
<i>rRhsValues</i>	Array of size <i>nRows</i> * <i>nRhs</i> . Contains real right-hand side vectors. Real data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <code>dss_create</code> .

cRhsValues

Array of size $nRows * nRhs$. Contains complex right-hand side vectors. Complex data, single or double precision as it is specified by the parameter *opt* in the routine `dss_create`.

RhsValues

Array of size $nRows * nRhs$. Contains right-hand side vectors. Real or complex data, single or double precision as it is specified by the parameter *opt* in the routine `dss_create`.

Output Parameters

rSolValues

Array of size $nCols * nRhs$. Contains real solution vectors. Real data, single or double precision as it is specified by the parameter *opt* in the routine `dss_create`.

cSolValues

Array of size $nCols * nRhs$. Contains complex solution vectors. Complex data, single or double precision as it is specified by the parameter *opt* in the routine `dss_create`.

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OUT_OF_MEMORY

MKL_DSS_DIAG_ERR

MKL_DSS_FAILURE

MKL_DSS_MSG_LVL_ERR

MKL_DSS_TERM_LVL_ERR

MKL_DSS_OOC_MEM_ERR

MKL_DSS_OOC_OC_ERR

MKL_DSS_OOC_RW_ERR

dss_delete

Deletes all of the data structures created during the solutions process.

Syntax

```
MKL_INT dss_delete(_MKL_DSS_HANDLE_t const *handle, MKL_INT const *opt)
```

Include Files

- `mkl.h`

Description

The routine `dss_delete` deletes all data structures created during the solving process.

Input Parameters

opt

Parameter to pass the DSS options. The default value is
`MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR`.

Output Parameters

handle Pointer to the data structure storing internal DSS results
 (`MKL_DSS_HANDLE`).

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_STATE_ERR`

`MKL_DSS_INVALID_OPTION`

`MKL_DSS_OUT_OF_MEMORY`

`MKL_DSS_MSG_LVL_ERR`

`MKL_DSS_TERM_LVL_ERR`

`dss_statistics`

Returns statistics about various phases of the solving process.

Syntax

```
MKL_INT dss_statistics(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, _CHARACTER_STR_t
const *statArr, _DOUBLE_PRECISION_t *retValues)
```

Include Files

- `mkl.h`

Description

The `dss_statistics` routine returns statistics about various phases of the solving process. This routine gathers the following statistics:

- time taken to do reordering,
- time taken to do factorization,
- duration of problem solving,
- determinant of the symmetric indefinite input matrix,
- inertia of the symmetric indefinite input matrix,
- number of floating point operations taken during factorization,
- total peak memory needed during the analysis and symbolic factorization,
- permanent memory needed from the analysis and symbolic factorization,
- memory consumption for the factorization and solve phases.

Statistics are returned in accordance with the input string specified by the parameter *statArr*. The value of the statistics is returned in double precision in a return array, which you must allocate beforehand.

For multiple statistics, multiple string constants separated by commas can be used as input. Return values are put into the return array in the same order as specified in the input string.

Statistics can only be requested at the appropriate stages of the solving process. For example, requesting `FactorTime` before a matrix is factored leads to an error.

The following table shows the point at which each individual statistics item can be requested:

Statistics Calling Sequences

Type of Statistics	When to Call
ReorderTime	After <code>dss_reorder</code> is completed successfully.
FactorTime	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.
SolveTime	After <code>dss_solve_real</code> or <code>dss_solve_complex</code> is completed successfully.
Determinant	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.
Inertia	After <code>dss_factor_real</code> is completed successfully and the matrix is real, symmetric, and indefinite.
Flops	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.
Peakmem	After <code>dss_reorder</code> is completed successfully.
Factormem	After <code>dss_reorder</code> is completed successfully.
Solvemem	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.

Input Parameters

<i>handle</i>	Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).										
<i>opt</i>	Parameter to pass the DSS options.										
<i>statArr</i>	Input string that defines the type of the returned statistics. The parameter can include one or more of the following string constants (case of the input string has no effect): <table> <tr> <td>ReorderTime</td><td>Amount of time taken to do the reordering.</td></tr> <tr> <td>FactorTime</td><td>Amount of time taken to do the factorization.</td></tr> <tr> <td>SolveTime</td><td>Amount of time taken to solve the problem after factorization.</td></tr> <tr> <td>Determinant</td><td>Determinant of the matrix <i>A</i>. For real matrices: the determinant is returned as <i>det_pow</i>, <i>det_base</i> in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_base}) < 10.0$ and $\text{determinant} = \text{det_base} * 10^{(\text{det_pow})}$. For complex matrices: the determinant is returned as <i>det_pow</i>, <i>det_re</i>, <i>det_im</i> in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_re}) + \text{abs}(\text{det_im}) < 10.0$ and $\text{determinant} = (\text{det_re}, \text{det_im}) * 10^{(\text{det_pow})}$.</td></tr> <tr> <td>Inertia</td><td>Inertia of a real symmetric matrix is defined as a triplet of nonnegative integers (p, n, z), where <i>p</i> is the number of positive eigenvalues, <i>n</i> is the number of negative eigenvalues, and <i>z</i> is the number of zero eigenvalues. <i>Inertia</i> is returned as three consecutive return array locations <i>p</i>, <i>n</i>, <i>z</i>.</td></tr> </table>	ReorderTime	Amount of time taken to do the reordering.	FactorTime	Amount of time taken to do the factorization.	SolveTime	Amount of time taken to solve the problem after factorization.	Determinant	Determinant of the matrix <i>A</i> . For real matrices: the determinant is returned as <i>det_pow</i> , <i>det_base</i> in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_base}) < 10.0$ and $\text{determinant} = \text{det_base} * 10^{(\text{det_pow})}$. For complex matrices: the determinant is returned as <i>det_pow</i> , <i>det_re</i> , <i>det_im</i> in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_re}) + \text{abs}(\text{det_im}) < 10.0$ and $\text{determinant} = (\text{det_re}, \text{det_im}) * 10^{(\text{det_pow})}$.	Inertia	Inertia of a real symmetric matrix is defined as a triplet of nonnegative integers (p, n, z) , where <i>p</i> is the number of positive eigenvalues, <i>n</i> is the number of negative eigenvalues, and <i>z</i> is the number of zero eigenvalues. <i>Inertia</i> is returned as three consecutive return array locations <i>p</i> , <i>n</i> , <i>z</i> .
ReorderTime	Amount of time taken to do the reordering.										
FactorTime	Amount of time taken to do the factorization.										
SolveTime	Amount of time taken to solve the problem after factorization.										
Determinant	Determinant of the matrix <i>A</i> . For real matrices: the determinant is returned as <i>det_pow</i> , <i>det_base</i> in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_base}) < 10.0$ and $\text{determinant} = \text{det_base} * 10^{(\text{det_pow})}$. For complex matrices: the determinant is returned as <i>det_pow</i> , <i>det_re</i> , <i>det_im</i> in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_re}) + \text{abs}(\text{det_im}) < 10.0$ and $\text{determinant} = (\text{det_re}, \text{det_im}) * 10^{(\text{det_pow})}$.										
Inertia	Inertia of a real symmetric matrix is defined as a triplet of nonnegative integers (p, n, z) , where <i>p</i> is the number of positive eigenvalues, <i>n</i> is the number of negative eigenvalues, and <i>z</i> is the number of zero eigenvalues. <i>Inertia</i> is returned as three consecutive return array locations <i>p</i> , <i>n</i> , <i>z</i> .										

Computing inertia can lead to incorrect results for matrixes with a cluster of eigenvalues which are near 0.

Inertia of a k -by- k real symmetric positive definite matrix is always $(k, 0, 0)$. Therefore `Inertia` is returned only in cases of real symmetric indefinite matrices. For all other matrix types, an error message is returned.

Flops

Number of floating point operations performed during the factorization.

Peakmem

Total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase.

Factormem

Permanent memory in kilobytes that the solver needs from the analysis and symbolic factorization phase in the factorization and solve phases.

Solvemem

Total double precision memory consumption (kilobytes) of the solver for the factorization and solve phases.

Output Parameters

`retValues`

Value of the statistics returned.

Finding 'time used to reorder' and 'inertia' of a matrix

The example below illustrates the use of the `dss_statistics` routine.

To find the above values, call `dss_statistics(handle, opt, statArr, retValue)`, where `statArr` is "ReorderTime,Inertia"

In this example, `retValue` has the following values:

<code>retValue[0]</code>	Time to reorder.
<code>retValue[1]</code>	Positive Eigenvalues.
<code>retValue[2]</code>	Negative Eigenvalues.
<code>retValue[3]</code>	Zero Eigenvalues.

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_INVALID_OPTION`

`MKL_DSS_STATISTICS_INVALID_MATRIX`

`MKL_DSS_STATISTICS_INVALID_STATE`

`MKL_DSS_STATISTICS_INVALID_STRING`

`MKL_DSS_MSG_LVL_ERR`

MKL_DSS_TERM_LVL_ERR

Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)

Intel® oneAPI Math Kernel Library (oneMKL) supports iterative sparse solvers (ISS) based on the reverse communication interface (RCI), referred to here as the RCI ISS interface. The RCI ISS interface implements a group of user-callable routines that are used in the step-by-step solving process of a symmetric positive definite system (RCI conjugate gradient solver, or RCI CG), and of a non-symmetric indefinite (non-degenerate) system (RCI flexible generalized minimal residual solver, or RCI FGMRES) of linear algebraic equations. This interface uses the general RCI scheme described in [Dong95].

See the [Appendix A Linear Solvers Basics](#) for discussion of terms and concepts related to the ISS routines.

The term *RCI* indicates that when the solver needs the results of certain operations (for example, matrix-vector multiplications), the user performs them and passes the result to the solver. This makes the solver more universal as it is independent of the specific implementation of the operations like the matrix-vector multiplication. To perform such operations, the user can use the built-in sparse matrix-vector multiplications and triangular solvers routines described in [Sparse BLAS Level 2 and Level 3 Routines](#).

NOTE

The RCI CG solver is implemented in two versions: for system of equations with a single right-hand side, and for systems of equations with multiple right-hand sides.

The CG method may fail to compute the solution or compute the wrong solution if the matrix of the system is not symmetric and not positive definite.

The FGMRES method may fail if the matrix is degenerate.

Table "RCI CG Interface Routines" lists the names of the routines, and describes their general use.

RCI ISS Interface Routines

Routine	Description
dcg_init , dcgmrhs_init , dfgmres_init	Initializes the solver.
dcg_check , dcgmrhs_check , dfgmres_check	Checks the consistency and correctness of the user defined data.
dcg , dcgmrhs , dfgmres	Computes the approximate solution vector.
dcg_get , dcgmrhs_get , dfgmres_get	Retrieves the number of the current iteration.

The Intel® oneAPI Math Kernel Library (oneMKL) RCI ISS interface routines are normally invoked in this order:

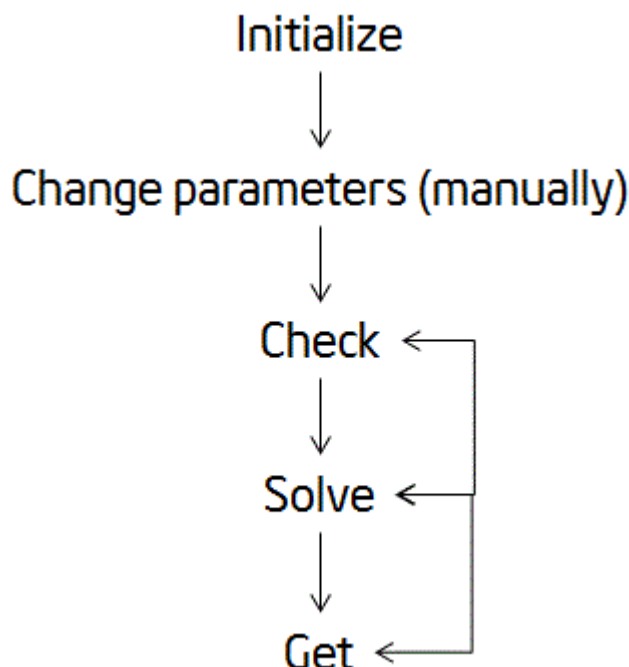
1. `<system_type>_init`
2. `<system_type>_check`
3. `<system_type>`
4. `<system_type>_get`

Advanced users can change that order if they need it. Others should follow the above order of calls.

The following diagram indicates the typical order in which the RCI ISS interface routines are invoked.

__border__ top

Typical Order for Invoking RCI ISS interface Routines



See the code examples that use the RCI ISS interface routines to solve systems of linear equations in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory.

- `examples/solverc/source`

CG Interface Description

Each routine for the RCI CG solver is implemented in two versions: for a system of equations with a single right-hand side (SRHS), and for a system of equations with multiple right-hand sides (MRHS). The names of routines for a system with MRHS contain the suffix `mrhs`.

Routine Options

All of the RCI CG routines have common parameters for passing various options to the routines (see [CG Common Parameters](#)). The values for these parameters can be changed during computations.

User Data Arrays

Many of the RCI CG routines take arrays of user data as input. For example, user arrays are passed to the routine `dcg` to compute the solution of a system of linear algebraic equations. The Intel® oneAPI Math Kernel Library (oneMKL) RCI CG routines do not make copies of the user input arrays to minimize storage requirements and improve overall run-time efficiency.

CG Common Parameters

NOTE

The default and initial values listed below are assigned to the parameters by calling the `dcg_init/dcgmrhs_init` routine.

<i>n</i>	MKL_INT, this parameter sets the size of the problem in the <code>dcg_init/dcgmrhs_init</code> routine. All the other routines use the <code>ipar[0]</code> parameter instead. Note that the coefficient matrix <i>A</i> is a square matrix of size $n \times n$.						
<i>x</i>	double array of size <i>n</i> for SRHS, or matrix of size $(n \times nrhs)$ for MRHS. This parameter contains the current approximation to the solution. Before the first call to the <code>dcg/dcgmrhs</code> routine, it contains the initial approximation to the solution.						
<i>nrhs</i>	MKL_INT, this parameter sets the number of right-hand sides for MRHS routines.						
<i>b</i>	double array containing a single right-hand side vector, or matrix of size $n \times nrhs$ containing right-hand side vectors.						
<i>RCI_request</i>	<p>MKL_INT, this parameter gives information about the result of work of the RCI CG routines. Negative values of the parameter indicate that the routine completed with errors or warnings. The 0 value indicates successful completion of the task. Positive values mean that you must perform specific actions:</p> <table> <tr> <td><i>RCI_request</i>= 1</td><td>multiply the matrix by <code>tmp [0:n - 1]</code>, put the result in <code>tmp[n:2*n - 1]</code>, and return the control to the <code>dcg/dcgmrhs</code> routine;</td></tr> <tr> <td><i>RCI_request</i>= 2</td><td>to perform the stopping tests. If they fail, return the control to the <code>dcg/dcgmrhs</code> routine. If the stopping tests succeed, it indicates that the solution is found and stored in the <i>x</i> array;</td></tr> <tr> <td><i>RCI_request</i>= 3</td><td> <p>for SRHS: apply the preconditioner to <code>tmp[2*n:3*n - 1]</code>, put the result in <code>tmp[3*n:4*n - 1]</code>, and return the control to the <code>dcg</code> routine;</p> <p>for MRHS: apply the preconditioner to <code>tmp[2+ipar[2]*n:(3 + ipar[2])*n - 1]</code>, put the result in <code>tmp[3*n:4*n - 1]</code>, and return the control to the <code>dcgmrhs</code> routine.</p> </td></tr> </table> <p>Note that the <code>dcg_get/dcgmrhs_get</code> routine does not change the parameter <i>RCI_request</i>. This enables use of this routine inside the <i>reverse communication</i> computations.</p>	<i>RCI_request</i> = 1	multiply the matrix by <code>tmp [0:n - 1]</code> , put the result in <code>tmp[n:2*n - 1]</code> , and return the control to the <code>dcg/dcgmrhs</code> routine;	<i>RCI_request</i> = 2	to perform the stopping tests. If they fail, return the control to the <code>dcg/dcgmrhs</code> routine. If the stopping tests succeed, it indicates that the solution is found and stored in the <i>x</i> array;	<i>RCI_request</i> = 3	<p>for SRHS: apply the preconditioner to <code>tmp[2*n:3*n - 1]</code>, put the result in <code>tmp[3*n:4*n - 1]</code>, and return the control to the <code>dcg</code> routine;</p> <p>for MRHS: apply the preconditioner to <code>tmp[2+ipar[2]*n:(3 + ipar[2])*n - 1]</code>, put the result in <code>tmp[3*n:4*n - 1]</code>, and return the control to the <code>dcgmrhs</code> routine.</p>
<i>RCI_request</i> = 1	multiply the matrix by <code>tmp [0:n - 1]</code> , put the result in <code>tmp[n:2*n - 1]</code> , and return the control to the <code>dcg/dcgmrhs</code> routine;						
<i>RCI_request</i> = 2	to perform the stopping tests. If they fail, return the control to the <code>dcg/dcgmrhs</code> routine. If the stopping tests succeed, it indicates that the solution is found and stored in the <i>x</i> array;						
<i>RCI_request</i> = 3	<p>for SRHS: apply the preconditioner to <code>tmp[2*n:3*n - 1]</code>, put the result in <code>tmp[3*n:4*n - 1]</code>, and return the control to the <code>dcg</code> routine;</p> <p>for MRHS: apply the preconditioner to <code>tmp[2+ipar[2]*n:(3 + ipar[2])*n - 1]</code>, put the result in <code>tmp[3*n:4*n - 1]</code>, and return the control to the <code>dcgmrhs</code> routine.</p>						
<i>ipar</i>	<p>MKL_INT array, of size 128 for SRHS, and of size $(128+2 \times nrhs)$ for MRHS. This parameter specifies the integer set of data for the RCI CG computations:</p> <table> <tr> <td><i>ipar</i>[0]</td><td>specifies the size of the problem. The <code>dcg_init/dcgmrhs_init</code> routine assigns <i>ipar</i>[0]=<i>n</i>. All the other routines use this parameter instead of <i>n</i>. There is no default value for this parameter.</td></tr> <tr> <td><i>ipar</i>[1]</td><td>specifies the type of output for error and warning messages generated by the RCI CG routines. The default value 6 means that all</td></tr> </table>	<i>ipar</i> [0]	specifies the size of the problem. The <code>dcg_init/dcgmrhs_init</code> routine assigns <i>ipar</i> [0]= <i>n</i> . All the other routines use this parameter instead of <i>n</i> . There is no default value for this parameter.	<i>ipar</i> [1]	specifies the type of output for error and warning messages generated by the RCI CG routines. The default value 6 means that all		
<i>ipar</i> [0]	specifies the size of the problem. The <code>dcg_init/dcgmrhs_init</code> routine assigns <i>ipar</i> [0]= <i>n</i> . All the other routines use this parameter instead of <i>n</i> . There is no default value for this parameter.						
<i>ipar</i> [1]	specifies the type of output for error and warning messages generated by the RCI CG routines. The default value 6 means that all						

messages are displayed on the screen. Otherwise, the error and warning messages are written to the newly created files `dcg_errors.txt` and `dcg_check_warnings.txt`, respectively. Note that if `ipar[5]` and `ipar[6]` parameters are set to 0, error and warning messages are not generated at all.

`ipar[2]`

for SRHS: contains the current stage of the RCI CG computations. The initial value is 1;

for MRHS: contains the number of the right-hand side for which the calculations are currently performed.

WARNING

Avoid altering this variable during computations.

`ipar[3]`

contains the current iteration number. The initial value is 0.

`ipar[4]`

specifies the maximum number of iterations. The default value is `min(150, n)`.

`ipar[5]`

if the value is not equal to 0, the routines output error messages in accordance with the parameter `ipar[1]`. Otherwise, the routines do not output error messages at all, but return a negative value of the parameter `RCI_request`. The default value is 1.

`ipar[6]`

if the value is not equal to 0, the routines output warning messages in accordance with the parameter `ipar[1]`. Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter `RCI_request`. The default value is 1.

`ipar[7]`

if the value is not equal to 0, the `dcg/dcgmrhs` routine performs the stopping test for the maximum number of iterations: $ipar[3] \leq ipar[4]$. Otherwise, the method is stopped and the corresponding value is assigned to the `RCI_request`. If the value is 0, the routine does not perform this stopping test. The default value is 1.

`ipar[8]`

if the value is not equal to 0, the `dcg/dcgmrhs` routine performs the residual stopping test: $dpar(5) \leq dpar(4) = dpar(1) * dpar(3) + dpar(2)$. Otherwise, the method is stopped and corresponding value is assigned to the

ipar[9]

RCI_request. If the value is 0, the routine does not perform this stopping test. The default value is 0.

if the value is not equal to 0, the *dcg/dcgmrhs* routine requests a user-defined stopping test by setting the output parameter *RCI_request*=2. If the value is 0, the routine does not perform the user defined stopping test. The default value is 1.

NOTE

At least one of the parameters *ipar*[7]-*ipar*[9] must be set to 1.

ipar[10]

if the value is equal to 0, the *dcg/dcgmrhs* routine runs the non-preconditioned version of the corresponding CG method. Otherwise, the routine runs the preconditioned version of the CG method, and by setting the output parameter *RCI_request*=3, indicates that you must perform the preconditioning step. The default value is 0.

ipar[11:127]

are reserved and not used in the current RCI CG SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array *ipar* with length 128 for a single right-hand side.

ipar[11:127 +
2**nrhs*]

are reserved for internal use in the current RCI CG SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array *ipar* with length 128+2**nrhs* for multiple right-hand sides.

dpar

double array, for SRHS of size 128, for MRHS of size (128+2**nrhs*); this parameter is used to specify the double precision set of data for the RCI CG computations, specifically:

dpar[0]

specifies the relative tolerance. The default value is 1.0×10^{-6} .

dpar[1]

specifies the absolute tolerance. The default value is 0.0.

<code>dpar[2]</code>	specifies the square norm of the initial residual (if it is computed in the <code>dcg/dcgmrhs</code> routine). The initial value is 0.0.
<code>dpar[3]</code>	service variable equal to $dpar[0]*dpar[2]+dpar[1]$ (if it is computed in the <code>dcg/dcgmrhs</code> routine). The initial value is 0.0.
<code>dpar[4]</code>	specifies the square norm of the current residual. The initial value is 0.0.
<code>dpar[5]</code>	specifies the square norm of residual from the previous iteration step (if available). The initial value is 0.0.
<code>dpar[6]</code>	contains the <i>alpha</i> parameter of the CG method. The initial value is 0.0.
<code>dpar[7]</code>	contains the <i>beta</i> parameter of the CG method, it is equal to $dpar[4]/dpar[5]$ The initial value is 0.0.
<code>dpar[8:127]</code>	are reserved and not used in the current RCI CG SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array `dpar` with length 128 for a single right-hand side.

<code>dpar(9:128+2*nrhs)</code> <code>[8:127 + 2*nrhs]</code>	are reserved for internal use in the current RCI CG SRHS and MRHS routines.
--	---

NOTE

For future compatibility, you must declare the array `dpar` with length $128+2*nrhs$ for multiple right-hand sides.

`tmp`

double array of size $(n*4)$ for SRHS, and $(n*(3+nrhs))$ for MRHS. This parameter is used to supply the double precision temporary space for the RCI CG computations, specifically:

<code>tmp[0:n - 1]</code>	specifies the current search direction. The initial value is 0.0.
<code>tmp[n:2*n - 1]</code>	contains the matrix multiplied by the current search direction. The initial value is 0.0.
<code>tmp[2*n:3*n - 1]</code>	contains the current residual. The initial value is 0.0.

<code>tmp[3*n:4*n - 1]</code>	contains the inverse of the preconditioner applied to the current residual for the SRHS version of CG. There is no initial value for this parameter.
<code>tmp[4*n:(4 + nrhs)*n - 1]</code>	contains the inverse of the preconditioner applied to the current residual for the MRHS version of CG. There is no initial value for this parameter.

NOTE

You can define this array in the code using RCI CG SRHS as `double tmp[3*n]` if you run only non-preconditioned CG iterations.

FGMRES Interface Description

Routine Options

All of the RCI FGMRES routines have common parameters for passing various options to the routines (see [FGMRES Common Parameters](#)). The values for these parameters can be changed during computations.

User Data Arrays

Many of the RCI FGMRES routines take arrays of user data as input. For example, user arrays are passed to the routine `dfgmresto` to compute the solution of a system of linear algebraic equations. To minimize storage requirements and improve overall run-time efficiency, the Intel® oneAPI Math Kernel Library (oneMKL) RCI FGMRES routines do not make copies of the user input arrays.

FGMRES Common Parameters

NOTE

The default and initial values listed below are assigned to the parameters by calling the `dfgmres_init` routine.

<code>n</code>	MKL_INT, this parameter sets the size of the problem in the <code>dfgmres_init</code> routine. All the other routines use the <code>ipar[0]</code> parameter instead. Note that the coefficient matrix <i>A</i> is a square matrix of size $n*n$.
<code>x</code>	double array, this parameter contains the current approximation to the solution vector. Before the first call to the <code>dfgmres</code> routine, it contains the initial approximation to the solution vector.
<code>b</code>	double array, this parameter contains the right-hand side vector. Depending on user requests (see the parameter <code>ipar[12]</code>), it might contain the approximate solution after execution.
<code>RCI_request</code>	MKL_INT, this parameter gives information about the result of work of the RCI FGMRES routines. Negative values of the parameter indicate that the routine completed with errors or warnings. The 0 value indicates successful completion of the task. Positive values mean that you must perform specific actions:

<code>RCI_request= 1</code>	multiply the matrix by <code>tmp[ipar[21] - 1:ipar[21] + n - 2]</code> , put the result in <code>tmp[ipar[22] - 1:ipar[22] + n - 2]</code> , and return the control to the <code>dfgmres</code> routine;
<code>RCI_request= 2</code>	perform the stopping tests. If they fail, return the control to the <code>dfgres</code> routine. Otherwise, the solution can be updated by a subsequent call to <code>dfgmres_get</code> routine;
<code>RCI_request= 3</code>	apply the preconditioner to <code>tmp[ipar[21] - 1:ipar[21] + n - 2]</code> , put the result in <code>tmp[ipar[22] - 1:ipar[22] + n - 2]</code> , and return the control to the <code>dfgmres</code> routine.
<code>RCI_request= 4</code>	check if the norm of the current orthogonal vector is zero, within the rounding or computational errors. Return the control to the <code>dfgmres</code> routine if it is not zero, otherwise complete the solution process by calling <code>dfgmres_get</code> routine.

`ipar[128]`

MKL_INT array, this parameter specifies the integer set of data for the RCI FGMRES computations:

<code>ipar[0]</code>	specifies the size of the problem. The <code>dfgmres_init</code> routine assigns <code>ipar[0]=n</code> . All the other routines uses this parameter instead of <code>n</code> . There is no default value for this parameter.
<code>ipar[1]</code>	specifies the type of output for error and warning messages that are generated by the RCI FGMRES routines. The default value 6 means that all messages are displayed on the screen. Otherwise the error and warning messages are written to the newly created file <code>MKL_RCI_FGMRES_Log.txt</code> . Note that if <code>ipar[5]</code> and <code>ipar[6]</code> parameters are set to 0, error and warning messages are not generated at all.
<code>ipar[2]</code>	contains the current stage of the RCI FGMRES computations. The initial value is 1.

WARNING

Avoid altering this variable during computations.

<code>ipar[3]</code>	contains the current iteration number. The initial value is 0.
<code>ipar[4]</code>	specifies the maximum number of iterations. The default value is <code>min(150,n)</code> .

<code>ipar[5]</code>	if the value is not 0, the routines output error messages in accordance with the parameter <code>ipar[1]</code> . If it is 0, the routines do not output error messages at all, but return a negative value of the parameter <code>RCI_request</code> . The default value is 1.
<code>ipar[6]</code>	if the value is not 0, the routines output warning messages in accordance with the parameter <code>ipar[1]</code> . Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter <code>RCI_request</code> . The default value is 1.
<code>ipar[7]</code>	if the value is not equal to 0, the <code>dfgmres</code> routine performs the stopping test for the maximum number of iterations: <code>ipar[3] ≤ ipar[4]</code> . If the value is 0, the <code>dfgmres</code> routine does not perform this stopping test. The default value is 1.
<code>ipar[8]</code>	if the value is not 0, the <code>dfgmres</code> routine performs the residual stopping test: <code>dpar[4] ≤ dpar[3]</code> . If the value is 0, the <code>dfgmres</code> routine does not perform this stopping test. The default value is 0.
<code>ipar[9]</code>	if the value is not 0, the <code>dfgmres</code> routine indicates that the user-defined stopping test should be performed by setting <code>RCI_request=2</code> . If the value is 0, the <code>dfgmres</code> routine does not perform the user-defined stopping test. The default value is 1.
<hr/> NOTE At least one of the parameters <code>ipar[7]</code> - <code>ipar[9]</code> must be set to 1. <hr/>	
<code>ipar[10]</code>	if the value is 0, the <code>dfgmres</code> routine runs the non-preconditioned version of the FGMRES method. Otherwise, the routine runs the preconditioned version of the FGMRES method, and requests that you perform the preconditioning step by setting the output parameter <code>RCI_request=3</code> . The default value is 0.
<code>ipar[11]</code>	if the value is not equal to 0, the <code>dfgmres</code> routine performs the automatic test for zero norm of the currently generated vector: <code>dpar[6] ≤ dpar[7]</code> , where <code>dpar[7]</code> contains the tolerance value. Otherwise, the routine indicates

that you must perform this check by setting the output parameter `RCI_request=4`. The default value is 0.

`ipar[12]`

if the value is equal to 0, the `dfgmres_get` routine updates the solution to the vector x according to the computations done by the `dfgmres` routine. If the value is positive, the routine writes the solution to the right-hand side vector b . If the value is negative, the routine returns only the number of the current iteration, and does not update the solution. The default value is 0.

NOTE

It is possible to call the `dfgmres_get` routine at any place in the code, but you must pay special attention to the parameter `ipar[12]`. The RCI FGMRES iterations can be continued after the call to `dfgmres_get` routine only if the parameter `ipar[12]` is not equal to zero. If `ipar[12]` is positive, then the updated solution overwrites the right-hand side in the vector b . If you want to run the restarted version of FGMRES with the same right-hand side, then it must be saved in a different memory location before the first call to the `dfgmres_get` routine with positive `ipar[12]`.

`ipar[13]`

contains the internal iteration counter that counts the number of iterations before the restart takes place. The initial value is 0.

WARNING

Do not alter this variable during computations.

`ipar[14]`

specifies the number of the non-restarted FGMRES iterations. To run the restarted version of the FGMRES method, assign the number of iterations to `ipar[14]` before the restart. The default value is $\min(150, n)$, which means that by default the non-restarted version of FGMRES method is used.

`ipar[15]`

service variable specifying the location of the rotated Hessenberg matrix from which the matrix stored in the packed format (see [Matrix Arguments](#) in the Appendix B for details) is started in the `tmp` array.

<code>ipar[16]</code>	service variable specifying the location of the rotation cosines from which the vector of cosines is started in the <code>tmp</code> array.
<code>ipar[17]</code>	service variable specifying the location of the rotation sines from which the vector of sines is started in the <code>tmp</code> array.
<code>ipar[18]</code>	service variable specifying the location of the rotated residual vector from which the vector is started in the <code>tmp</code> array.
<code>ipar[19]</code>	service variable, specifies the location of the least squares solution vector from which the vector is started in the <code>tmp</code> array.
<code>ipar[20]</code>	service variable specifying the location of the set of preconditioned vectors from which the set is started in the <code>tmp</code> array. The memory locations in the <code>tmp</code> array starting from <code>ipar[20]</code> are used only for the preconditioned FGMRES method.
<code>ipar[21]</code>	specifies the memory location from which the first vector (source) used in operations requested via <code>RCI_request</code> is started in the <code>tmp</code> array.
<code>ipar[22]</code>	specifies the memory location from which the second vector (output) used in operations requested via <code>RCI_request</code> is started in the <code>tmp</code> array.
<code>ipar[23:127]</code>	are reserved and not used in the current RCI FGMRES routines.

NOTE

You must declare the array `ipar` with length 128. While defining the array in the code as `ipar[23]` works, there is no guarantee of future compatibility with Intel® oneAPI Math Kernel Library (oneMKL).

`dpar(128)` double array, this parameter specifies the double precision set of data for the RCI CG computations, specifically:

<code>dpar[0]</code>	specifies the relative tolerance. The default value is 1.0e-6.
<code>dpar[1]</code>	specifies the absolute tolerance. The default value is 0.0e-0.
<code>dpar[2]</code>	specifies the Euclidean norm of the initial residual (if it is computed in the <code>dfgmres</code> routine). The initial value is 0.0.

<code>dpar[3]</code>	service variable equal to $dpar[0] * dpar[2] + dpar[1]$ (if it is computed in the <code>dfgmres</code> routine). The initial value is 0.0.
<code>dpar[4]</code>	specifies the Euclidean norm of the current residual. The initial value is 0.0.
<code>dpar[5]</code>	specifies the Euclidean norm of residual from the previous iteration step (if available). The initial value is 0.0.
<code>dpar[6]</code>	contains the norm of the generated vector. The initial value is 0.0.

NOTE

In terms of [Saad03] this parameter is the coefficient $h_{k+1,k}$ of the Hessenberg matrix.

<code>dpar[7]</code>	contains the tolerance for the zero norm of the currently generated vector. The default value is 1.0e-12.
<code>dpar[8:127]</code>	are reserved and not used in the current RCI FGMRES routines.

NOTE

You must declare the array `dpar` with length 128. While defining the array in the code as `double dpar[8]` works, there is no guarantee of future compatibility with Intel® oneAPI Math Kernel Library (oneMKL).

`tmp`

double array of size $((2 * ipar[14] + 1) * n + ipar[14] * (ipar[14] + 9) / 2 + 1)$ used to supply the double precision temporary space for the RCI FGMRES computations, specifically:

<code>tmp[0:ipar[15] - 2]</code>	contains the sequence of vectors generated by the FGMRES method. The initial value is 0.0.
<code>tmp[ipar[15] - 1:ipar[16] - 2]</code>	contains the rotated Hessenberg matrix generated by the FGMRES method; the matrix is stored in the packed format. There is no initial value for this part of <code>tmp</code> array.
<code>tmp[ipar[16] - 1:ipar[17] - 2]</code>	contains the rotation cosines vector generated by the FGMRES method. There is no initial value for this part of <code>tmp</code> array.
<code>tmp[ipar[17] - 1:ipar[18] - 2]</code>	contains the rotation sines vector generated by the FGMRES method. There is no initial value for this part of <code>tmp</code> array.

```
tmp[ipar[18] -
1:ipar[19] - 2]
```

contains the rotated residual vector generated by the FGMRES method. There is no initial value for this part of *tmp* array.

```
tmp[ipar[19] -
1:ipar[20] - 2]
```

contains the solution vector to the least squares problem generated by the FGMRES method. There is no initial value for this part of *tmp* array.

```
tmp[ipar[20] - 1:*
```

contains the set of preconditioned vectors generated for the FGMRES method by the user. This part of *tmp* array is not used if the non-preconditioned version of FGMRES method is called. There is no initial value for this part of *tmp* array.

NOTE

You can define this array in the code as `double tmp[(2*ipar[14] + 1)*n + ipar[14]*(ipar[14] + 9)/2 + 1]` if you run only non-preconditioned FGMRES iterations.

RCI ISS Routines

dcg_init

Initializes the solver.

Syntax

```
void dcg_init (const MKL_INT *n , const double *x , const double *b , MKL_INT
*RCI_request , MKL_INT *ipar , double *dpar , double *tmp );
```

Include Files

- mkl.h

Description

The routine `dcg_init` initializes the solver. After initialization, all subsequent invocations of the Intel® oneAPI Math Kernel Library (oneMKL) RCI CG routines use the values of all parameters returned by the routine `dcg_init`. Advanced users can skip this step and set the values in the *ipar* and *dpar* arrays directly.

Caution

You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dcg_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

n	Sets the size of the problem.
x	Array of size n . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to b .
b	Array of size n . Contains the right-hand side vector.

Output Parameters

$RCI_request$	Gives information about the result of the routine.
$ipar$	Array of size 128. Refer to the CG Common Parameters .
$dpar$	Array of size 128. Refer to the CG Common Parameters .
tmp	Array of size $(n*4)$. Refer to the CG Common Parameters .

Return Values

$RCI_request = 0$	Indicates that the task completed normally.
$RCI_request = -10000$	Indicates failure to complete the task.

dcg_check

Checks consistency and correctness of the user defined data.

Syntax

```
void dcg_check (const MKL_INT *n , const double *x , const double *b , MKL_INT
*RCI_request , MKL_INT *ipar , double *dpar , double *tmp );
```

Include Files

- `mkl.h`

Description

The routine `dcg_check` checks consistency and correctness of the parameters to be passed to the solver routine `dcg`. However this operation does not guarantee that the solver returns the correct result. It only reduces the chance of making a mistake in the parameters of the method. Skip this operation only if you are sure that the correct data is specified in the solver parameters.

The lengths of all vectors must be defined in a previous call to the `dcg_init` routine.

If none of the stopping criteria ($ipar[7]-ipar[9]$) has been enabled, both $ipar[7]$ and $ipar[8]$ will be set to 1.

Input Parameters

$ipar$	Array of size 128. Refer to the FGMRES Common Parameters .
n	Sets the size of the problem.
x	Array of size n . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to b .

b Array of size *n*. Contains the right-hand side vector.

Output Parameters

RCI_request Gives information about result of the routine.

ipar Array of size 128. Refer to the [CG Common Parameters](#). Only *ipar*[7]-*ipar*[8] might be changed

dpar Array of size 128. Refer to the [CG Common Parameters](#).

tmp Array of size (*n**4). Refer to the [CG Common Parameters](#).

Return Values

RCI_request = 0 Indicates that the task completed normally.

RCI_request = -1100 Indicates that the task is interrupted and the errors occur.

RCI_request = -1001 Indicates that there are some warning messages.

RCI_request = -1010 Indicates that the routine changed some parameters to make them consistent or correct.

RCI_request = -1011 Indicates that there are some warning messages and that the routine changed some parameters.

dcg

Computes the approximate solution vector.

Syntax

```
void dcg (const MKL_INT *n , double *x , const double *b , MKL_INT *RCI_request ,
MKL_INT *ipar , double *dpar , double *tmp );
```

Include Files

- `mkl.h`

Description

The `dcg` routine computes the approximate solution vector using the CG method [Young71]. The routine `dcg` uses the vector in the array *x* before the first call as an initial approximation to the solution. The parameter *RCI_request* gives you information about the task completion and requests results of certain operations that are required by the solver.

Note that lengths of all vectors must be defined in a previous call to the `dcg_init` routine.

Input Parameters

n Sets the size of the problem.

x Array of size *n*. Contains the initial approximation to the solution vector.

b Array of size *n*. Contains the right-hand side vector.

tmp Array of size (*n**4). Refer to the [CG Common Parameters](#).

Output Parameters

<code>RCI_request</code>	Gives information about result of work of the routine.
<code>x</code>	Array of size n . Contains the updated approximation to the solution vector.
<code>ipar</code>	Array of size 128. Refer to the CG Common Parameters .
<code>dpar</code>	Array of size 128. Refer to the CG Common Parameters .
<code>tmp</code>	Array of size $(n*4)$. Refer to the CG Common Parameters .

Return Values

<code>RCI_request=0</code>	Indicates that the task completed normally and the solution is found and stored in the vector <code>x</code> . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the <code>RCI_request= 2</code> .
<code>RCI_request=-1</code>	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if you request both tests.
<code>RCI_request=-2</code>	Indicates that the routine was interrupted because of an attempt to divide by zero. This situation happens if the matrix is non-positive definite or almost non-positive definite.
<code>RCI_request=- 10</code>	Indicates that the routine was interrupted because the residual norm is invalid. This usually happens because the value <code>dpar[5]</code> was altered outside of the routine, or the <code>dcg_check</code> routine was not called.
<code>RCI_request=-11</code>	Indicates that the routine was interrupted because it enters the infinite cycle. This usually happens because the values <code>ipar[7]</code> , <code>ipar[8]</code> , <code>ipar[9]</code> were altered outside of the routine, or the <code>dcg_check</code> routine was not called.
<code>RCI_request= 1</code>	Indicates that you must multiply the matrix by <code>tmp[0:n - 1]</code> , put the result in the <code>tmp[n:2*n - 1]</code> , and return control back to the routine <code>dcg</code> .
<code>RCI_request= 2</code>	Indicates that you must perform the stopping tests. If they fail, return control back to the <code>dcg</code> routine. Otherwise, the solution is found and stored in the vector <code>x</code> .
<code>RCI_request= 3</code>	Indicates that you must apply the preconditioner to <code>[2*n:3*n - 1]</code> , put the result in the <code>[3*n:4*n - 1]</code> , and return control back to the routine <code>dcg</code> .

`dcg_get`

Retrieves the number of the current iteration.

Syntax

```
void dcg_get (const MKL_INT *n , const double *x , const double *b , const MKL_INT
*RCI_request , const MKL_INT *ipar , const double *dpar , const double *tmp , MKL_INT
*itercount );
```

Include Files

- mkl.h

Description

The routine `dcg_get` retrieves the current iteration number of the solutions process.

Input Parameters

<i>n</i>	Sets the size of the problem.
<i>x</i>	Array of size <i>n</i> . Contains the approximation vector to the solution.
<i>b</i>	Array of size <i>n</i> . Contains the right-hand side vector.
<i>RCI_request</i>	This parameter is not used.
<i>ipar</i>	Array of size 128. Refer to the CG Common Parameters .
<i>dpar</i>	Array of size 128. Refer to the CG Common Parameters .
<i>tmp</i>	Array of size $(n*4)$. Refer to the CG Common Parameters .

Output Parameters

<i>itercount</i>	Returns the current iteration number.
------------------	---------------------------------------

Return Values

The routine `dcg_get` has no return values.

dcgmrhs_init

Initializes the RCI CG solver with MHRS.

Syntax

```
void dcgmrhs_init (const MKL_INT *n , const double *x , const MKL_INT *nrhs , const
double *b , const MKL_INT *method , MKL_INT *RCI_request , MKL_INT *ipar , double
*dpar , double *tmp );
```

Include Files

- mkl.h

Description

The routine `dcgmrhs_init` initializes the solver. After initialization all subsequent invocations of the Intel® oneAPI Math Kernel Library (oneMKL) RCI CG with multiple right-hand sides (MRHS) routines use the values of all parameters that are returned by `dcgmrhs_init`. Advanced users may skip this step and set the values to these parameters directly in the appropriate routines.

WARNING

You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dcgmrhs_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

<i>n</i>	Sets the size of the problem.
<i>x</i>	Array of size $n*nrhs$. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to <i>b</i> .
<i>nrhs</i>	Sets the number of right-hand sides.
<i>b</i>	Array of size $n*nrhs$. Contains the right-hand side vectors.
<i>method</i>	Specifies the method of solution: A value of 1 indicates CG with multiple right-hand sides (default value)

Output Parameters

<i>RCI_request</i>	Gives information about the result of the routine.
<i>ipar</i>	Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>dpar</i>	Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	Array of size $(n*(3+nrhs))$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -10000	Indicates failure to complete the task.

`dcgmrhs_check`

Checks consistency and correctness of the user defined data.

Syntax

```
void dcgmrhs_check (const MKL_INT *n , const double *x , const MKL_INT *nrhs , const
double *b , MKL_INT *RCI_request , MKL_INT *ipar , double *dpar , double *tmp );
```

Include Files

- `mkl.h`

Description

The routine `dcgmrhs_check` checks the consistency and correctness of the parameters to be passed to the solver routine `dcgmrhs`. While this operation reduces the chance of making a mistake in the parameters, it does not guarantee that the solver returns the correct result.

If you are sure that the correct data is *specified* in the solver parameters, you can skip this operation.

The lengths of all vectors must be defined in a previous call to the `dcgmrhs_init` routine.

If none of the stopping criteria ($ipar[7]-ipar[9]$) has been enabled, both $ipar[7]$ and $ipar[8]$ will be set to 1.

Input Parameters

n	Sets the size of the problem.
x	Array of size $n*nrhs$. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to b .
$nrhs$	This parameter sets the number of right-hand sides.
b	Array of size $n*nrhs$. Contains the right-hand side vectors.

Output Parameters

$RCI_request$	Returns information about the results of the routine.
$ipar$	Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters . Only $ipar[7]-ipar[8]$ might be changed.
$dpar$	Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
tmp	Array of size $(n*(3+nrhs))$. Refer to the CG Common Parameters .

Return Values

$RCI_request = 0$	Indicates that the task completed normally.
$RCI_request = -1100$	Indicates that the task is interrupted and the errors occur.
$RCI_request = -1001$	Indicates that there are some warning messages.
$RCI_request = -1010$	Indicates that the routine changed some parameters to make them consistent or correct.
$RCI_request = -1011$	Indicates that there are some warning messages and that the routine changed some parameters.

dcgmrhs

Computes the approximate solution vectors.

Syntax

```
void dcgmrhs (const MKL_INT *n , double *x , const MKL_INT *nrhs , const double *b ,
MKL_INT *RCI_request , MKL_INT *ipar , double *dpar , double *tmp );
```

Include Files

- `mkl.h`

Description

The routine `dcgmrhs` computes approximate solution vectors using the CG with multiple right-hand sides (MRHS) method [Young71]. The routine `dcgmrhs` uses the value that was in the x before the first call as an initial approximation to the solution. The parameter $RCI_request$ gives information about task completion status and requests results of certain operations that are required by the solver.

Note that lengths of all vectors are assumed to have been defined in a previous call to the `dcgmrhs_init` routine.

Input Parameters

<i>n</i>	Sets the size of the problem, and the sizes of arrays <i>x</i> and <i>b</i> .
<i>x</i>	Array of size $n \cdot nrhs$. Contains the initial approximation to the solution vectors.
<i>nrhs</i>	Sets the number of right-hand sides.
<i>b</i>	Array of size $n \cdot nrhs$. Contains the right-hand side vectors.
<i>tmp</i>	Array of size $(n, 3 + nrhs)$. Refer to the CG Common Parameters .

Output Parameters

<i>RCI_request</i>	Gives information about result of work of the routine.
<i>x</i>	Array of size $(n \cdot nrhs)$. Contains the updated approximation to the solution vectors.
<i>ipar</i>	Array of size $(128 + 2 \cdot nrhs)$. Refer to the CG Common Parameters .
<i>dpar</i>	Array of size $(128 + 2 \cdot nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	Array of size $(n \cdot (3 + nrhs))$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> =0	Indicates that the task completed normally and the solution is found and stored in the vector <i>x</i> . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the <i>RCI_request</i> = 2.
<i>RCI_request</i> =-1	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if both tests are requested by the user.
<i>RCI_request</i> =-2	The routine was interrupted because of an attempt to divide by zero. This situation happens if the matrix is non-positive definite or almost non-positive definite.
<i>RCI_request</i> =- 10	Indicates that the routine was interrupted because the residual norm is invalid. This usually happens because the value <i>dpar</i> [5] was altered outside of the routine, or the <i>dcg_check</i> routine was not called.
<i>RCI_request</i> =-11	Indicates that the routine was interrupted because it enters the infinite cycle. This usually happens because the values <i>ipar</i> [7], <i>ipar</i> [8], <i>ipar</i> [9] were altered outside of the routine, or the <i>dcg_check</i> routine was not called.
<i>RCI_request</i> = 1	Indicates that you must multiply the matrix by <i>tmp</i> [0: <i>n</i> - 1], put the result in the <i>tmp</i> [<i>n</i> :2* <i>n</i> - 1], and return control back to the routine <i>dcg</i> .

`RCI_request= 2`

Indicates that you must perform the stopping tests. If they fail, return control back to the `dcg` routine. Otherwise, the solution is found and stored in the vector `x`.

`RCI_request= 3`

Indicates that you must apply the preconditioner to `tmp[2*n:3*n - 1]`, put the result in the `tmp[3*n:4*n - 1]`, and return control back to the routine `dcg`.

dcgmrhs_get

Retrieves the number of the current iteration.

Syntax

```
void dcgmrhs_get (const MKL_INT *n , const double *x , const MKL_INT *nrhs , const
double *b , const MKL_INT *RCI_request , const MKL_INT *ipar , const double *dpar ,
const double *tmp , MKL_INT *itercount );
```

Include Files

- `mkl.h`

Description

The routine `dcgmrhs_get` retrieves the current iteration number of the solving process.

Input Parameters

<code>n</code>	Sets the size of the problem.
<code>x</code>	Array of size $n*nrhs$. Contains the initial approximation to the solution vectors.
<code>nrhs</code>	Sets the number of right-hand sides.
<code>b</code>	Array of size $n*nrhs$. Contains the right-hand side .
<code>RCI_request</code>	This parameter is not used.
<code>ipar</code>	Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<code>dpar</code>	Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<code>tmp</code>	Array of size $(n*(3+nrhs))$. Refer to the CG Common Parameters .

Output Parameters

<code>itercount</code>	Array of size $nrhs$. Returns the current iteration number for each right-hand side.
------------------------	---

Return Values

The routine `dcgmrhs_get` has no return values.

dfgmres_init

Initializes the solver.

Syntax

```
void dfgmres_init (const MKL_INT *n , const double *x , const double *b , MKL_INT
*RCI_request , MKL_INT *ipar , double *dpar , double *tmp );
```

Include Files

- mkl.h

Description

The routine `dfgmres_init` initializes the solver. After initialization all subsequent invocations of Intel® oneAPI Math Kernel Library (oneMKL) RCI FGMRES routines use the values of all parameters that are returned by `dfgmres_init`. Advanced users can skip this step and set the values in the `ipar` and `dpar` arrays directly.

WARNING

You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dfgmres_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

<i>n</i>	Sets the size of the problem.
<i>x</i>	Array of size <i>n</i> . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to <i>b</i> .
<i>b</i>	Array of size <i>n</i> . Contains the right-hand side vector.

Output Parameters

<i>RCI_request</i>	Gives information about the result of the routine.
<i>ipar</i>	Array of size 128. Refer to the FGMRES Common Parameters .
<i>dpar</i>	Array of size 128. Refer to the FGMRES Common Parameters .
<i>tmp</i>	Array of size $((2 * ipar[14] + 1) * n + ipar[14] * (ipar[14] + 9) / 2 + 1)$. Refer to the FGMRES Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -10000	Indicates failure to complete the task.

dfgmres_check

Checks consistency and correctness of the user defined data.

Syntax

```
void dfgmres_check (const MKL_INT *n, const double *x, const double *b, MKL_INT
*RCI_request, MKL_INT *ipar, double *dpar, double *tmp );
```

Include Files

- `mkl.h`

Description

The routine `dfgmres_check` checks consistency and correctness of the parameters to be passed to the solver routine `dfgmres`. However, this operation does not guarantee that the method gives the correct result. It only reduces the chance of making a mistake in the parameters of the routine. Skip this operation only if you are sure that the correct data is specified in the solver parameters.

The lengths of all vectors are assumed to have been defined in a previous call to the `dfgmres_init` routine.

In particular, the routine checks the consistency of `ipar[15]-ipar[20]` and `ipar[0], ipar[14]`. If the values do not agree, the routine emits a warning and modifies `ipar[15]-ipar[20]` to comply with the values of `ipar[0], ipar[14]`. A possible use case for this modification is a non-default value (not the one set by a possible call to `dfgmres_init`) of `ipar[14]`.

Also, if none of the stopping criteria (`ipar[7]-ipar[9]`) has been enabled, both `ipar[7]` and `ipar[9]` will be set to 1.

NOTE: It is not strictly necessary to call the `dfgmres_check` routine unless the values of `ipar[14]` or `ipar[0]` are changed after the last call to `dfgmres_init`.

Input Parameters

<code>ipar</code>	Array of size 128. Refer to the FGMRES Common Parameters .
<code>n</code>	Sets the size of the problem.
<code>x</code>	Array of size <code>n</code> . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to <code>b</code> .
<code>b</code>	Array of size <code>n</code> . Contains the right-hand side vector.

Output Parameters

<code>RCI_request</code>	Gives information about result of the routine.
<code>ipar</code>	Array of size 128. Refer to the FGMRES Common Parameters . Only <code>ipar[7]-ipar[8]</code> and <code>ipar[15]-ipar[20]</code> might be changed.
<code>dpar</code>	Array of size 128. Refer to the FGMRES Common Parameters .
<code>tmp</code>	Array of size $((2 * ipar[14] + 1) * n + ipar[14] * (ipar[14] + 9) / 2 + 1)$. Refer to the FGMRES Common Parameters .

Return Values

<code>RCI_request = 0</code>	Indicates that the task completed normally.
<code>RCI_request = -1100</code>	Indicates that the task is interrupted and the errors occur.
<code>RCI_request = -1001</code>	Indicates that there are some warning messages.
<code>RCI_request = -1010</code>	Indicates that the routine changed some parameters to make them consistent or correct.
<code>RCI_request = -1011</code>	Indicates that there are some warning messages and that the routine changed some parameters.

dfgmres*Makes the FGMRES iterations.***Syntax**

```
void dfgmres (const MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, MKL_INT
*ipar, double *dpar, double *tmp );
```

Include Files

- mkl.h

Description

The routine `dfgmres` performs the FGMRES iterations [Saad03], using the value that was in the array `x` before the first call as an initial approximation of the solution vector. To update the current approximation to the solution, the `dfgmres_get` routine must be called. The RCI FGMRES iterations can be continued after the call to the `dfgmres_get` routine only if the value of the parameter `ipar[12]` is not equal to 0 (default value). Note that the updated solution overwrites the right-hand side in the vector `b` if the parameter `ipar[12]` is positive, and the restarted version of the FGMRES method can not be run. If you want to keep the right-hand side, you must save it in a different memory location before the first call to the `dfgmres_get` routine with a positive `ipar[12]`.

The parameter `RCI_request` gives information about the task completion and requests results of certain operations that the solver requires.

The lengths of all the vectors must be defined in a previous call to the `dfgmres_init` routine.

Input Parameters

<code>n</code>	Sets the size of the problem.
<code>x</code>	Array of size <code>n</code> . Contains the initial approximation to the solution vector.
<code>b</code>	Array of size <code>n</code> . Contains the right-hand side vector.
<code>tmp</code>	Array of size [12]. Refer to the FGMRES Common Parameters .

Output Parameters

<code>RCI_request</code>	Informs about result of work of the routine.
<code>ipar</code>	Array of size 128. Refer to the FGMRES Common Parameters .
<code>dpar</code>	Array of size 128. Refer to the FGMRES Common Parameters .
<code>tmp</code>	Array of size $((2*ipar[14]+1)*n+ipar[14]*ipar[14]+9)/2 + 1$. Refer to the FGMRES Common Parameters .

Return Values

<code>RCI_request=0</code>	Indicates that the task completed normally and the solution is found and stored in the vector <code>x</code> . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the <code>RCI_request= 2</code> or 4.
----------------------------	--

`RCI_request=-1`

Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if you request both tests.

`RCI_request= -10`

Indicates that the routine was interrupted because of an attempt to divide by zero. Usually this happens if the matrix is degenerate or almost degenerate. However, it may happen if the parameter *dpar* is altered, or if the method is not stopped when the solution is found.

`RCI_request= -11`

Indicates that the routine was interrupted because it entered an infinite cycle. Usually this happens because the values *ipar*[7], *ipar*[8], *ipar*[9] were altered outside of the routine, or the *dfgmres_check* routine was not called.

`RCI_request= -12`

Indicates that the routine was interrupted because errors were found in the method parameters. Usually this happens if the parameters *ipar* and *dpar* were altered by mistake outside the routine.

`RCI_request= 1`

Indicates that you must multiply the matrix by *tmp*[*ipar*[21] - 1:*ipar*[21] + *n* - 2], put the result in the *tmp*[*ipar*[22] - 1:*ipar*[22] + *n* - 2], and return control back to the routine *dfgmres*.

`RCI_request= 2`

Indicates that you must perform the stopping tests. If they fail, return control to the *dfgmres* routine. Otherwise, the FGMRES solution is found, and you can run the *fgmres_get* routine to update the computed solution in the vector *x*.

`RCI_request= 3`

Indicates that you must apply the inverse preconditioner to *tmp*[*ipar*[21] - 1:*ipar*[21] + *n* - 2], put the result in the *tmp*[*ipar*[22] - 1:*ipar*[22] + *n* - 2], and return control back to the routine *dfgmres*.

`RCI_request= 4`

Indicates that you must check the norm of the currently generated vector. If it is not zero within the computational/rounding errors, return control to the *dfgmres* routine. Otherwise, the FGMRES solution is found, and you can run the *dfgmres_get* routine to update the computed solution in the vector *x*.

dfgmres_get

Retrieves the number of the current iteration and updates the solution.

Syntax

```
void dfgmres_get (const MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, const
MKL_INT *ipar, const double *dpar, double *tmp, MKL_INT *itercount );
```

Include Files

- `mkl.h`

Description

The routine `dfgmres_get` retrieves the current iteration number of the solution process and updates the solution according to the computations performed by the `dfgmres` routine. To retrieve the current iteration number only, set the parameter `ipar[12] = -1` beforehand. Normally, you should do this before proceeding further with the computations. If the intermediate solution is needed, the method parameters must be set properly. For details see [FGMRES Common Parameters](#) and the Iterative Sparse Solver code examples in the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- `examples/solverc/source`

Input Parameters

<code>n</code>	Sets the size of the problem.
<code>ipar</code>	Array of size 128. Refer to the FGMRES Common Parameters .
<code>dpar</code>	Array of size 128. Refer to the FGMRES Common Parameters .
<code>tmp</code>	Array of size $((2 * ipar[14] + 1) * n + ipar[14] * ipar[14] + 9) / 2 + 1$. Refer to the FGMRES Common Parameters .

Output Parameters

<code>x</code>	Array of size <code>n</code> . If <code>ipar[12] = 0</code> , it contains the updated approximation to the solution according to the computations done in <code>dfgmres</code> routine. Otherwise, it is not changed.
<code>b</code>	Array of size <code>n</code> . If <code>ipar(13) > 0</code> , it contains the updated approximation to the solution according to the computations done in <code>dfgmres</code> routine. Otherwise, it is not changed.
<code>RCI_request</code>	Gives information about result of the routine.
<code>itercount</code>	Contains the value of the current iteration number.

Return Values

<code>RCI_request = 0</code>	Indicates that the task completed normally.
<code>RCI_request = -12</code>	Indicates that the routine was interrupted because errors were found in the routine parameters. Usually this happens if the parameters <code>ipar</code> and <code>dpar</code> were altered by mistake outside of the routine.
<code>RCI_request = -10000</code>	Indicates that the routine failed to complete the task.

RCI ISS Implementation Details

Several aspects of the Intel® oneAPI Math Kernel Library (oneMKL) RCI ISS interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, include one of the Intel® oneAPI Math Kernel Library (oneMKL) RCI ISS language-specific header files.

NOTE

Intel® oneAPI Math Kernel Library (oneMKL) does not support the RCI ISS interface unless you include the language-specific header file.

Preconditioners based on Incomplete LU Factorization Technique

Preconditioners, or accelerators are used to accelerate an iterative solution process. In some cases, their use can reduce the number of iterations dramatically and thus lead to better solver performance. Although the terms *preconditioner* and *accelerator* are synonyms, hereafter only *preconditioner* is used.

Intel® oneAPI Math Kernel Library (oneMKL) provides two preconditioners, ILU0 and ILUT, for sparse matrices presented in the format accepted in the Intel® oneAPI Math Kernel Library (oneMKL) direct sparse solvers (three-array variation of the CSR storage format described in [Sparse Matrix Storage Format](#)). The algorithms used are described in [Saad03].

The ILU0 preconditioner is based on a well-known factorization of the original matrix into a product of two triangular matrices: lower and upper triangular matrices. Usually, such decomposition leads to some fill-in in the resulting matrix structure in comparison with the original matrix. The distinctive feature of the ILU0 preconditioner is that it preserves the structure of the original matrix in the result.

Unlike the ILU0 preconditioner, the ILUT preconditioner preserves some resulting fill-in in the preconditioner matrix structure. The distinctive feature of the ILUT algorithm is that it calculates each element of the preconditioner and saves each one if it satisfies two conditions simultaneously: its value is greater than the product of the given tolerance and matrix row norm, and its value is in the given bandwidth of the resulting preconditioner matrix.

Both ILU0 and ILUT preconditioners can apply to any non-degenerate matrix. They can be used alone or together with the Intel® oneAPI Math Kernel Library (oneMKL) RCI FGMRES solver (see [Sparse Solver Routines](#)). Avoid using these preconditioners with MKL RCI CG solver because in general, they produce a non-symmetric resulting matrix even if the original matrix is symmetric. Usually, an inverse of the preconditioner is required in this case. To do this the Intel® oneAPI Math Kernel Library (oneMKL) triangular solver routine `mkl_dcsrtrsv` must be applied twice: for the lower triangular part of the preconditioner, and then for its upper triangular part.

NOTE

Although ILU0 and ILUT preconditioners apply to any non-degenerate matrix, in some cases the algorithm may fail to ensure successful termination and the required result. Whether or not the preconditioner produces an acceptable result can only be determined in practice.

A preconditioner may increase the number of iterations for an arbitrary case of the system and the initial solution, and even ruin the convergence. It is your responsibility as a user to choose a suitable preconditioner.

General Scheme of Using ILUT and RCI FGMRES Routines

The general scheme for use is the same for both preconditioners. Some differences exist in the calling parameters of the preconditioners and in the subsequent call of two triangular solvers. You can see all these differences in the preconditioner code examples (`dcsrilu*.*`) in the `examples` folder of the Intel® oneAPI Math Kernel Library (oneMKL) installation directory:

- `examples/solverc/source`

ILU0 and ILUT Preconditioners Interface Description

The concepts required to understand the use of the Intel® oneAPI Math Kernel Library (oneMKL) preconditioner routines are discussed in the [Appendix A Linear Solvers Basics](#).

User Data Arrays

The preconditioner routines take arrays of user data as input. To minimize storage requirements and improve overall run-time efficiency, the Intel® oneAPI Math Kernel Library (oneMKL) preconditioner routines do not make copies of the user input arrays.

Common Parameters

Some parameters of the preconditioners are common with the [FGMRES Common Parameters](#). The routine `dfgmres_init` specifies their default and initial values. However, some parameters can be redefined with other values. These parameters are listed below.

For the ILU0 preconditioner:

`ipar[1]` - specifies the destination of error messages generated by the ILU0 routine. The default value 6 means that all error messages are displayed on the screen. Otherwise routine creates a log file called `MKL_PREC_log.txt` and writes error messages to it. Note if the parameter `ipar[5]` is set to 0, then error messages are not generated at all.

`ipar[5]` - specifies whether error messages are generated. If its value is not equal to 0, the ILU0 routine returns error messages as specified by the parameter `ipar[1]`. Otherwise, the routine does not generate error messages at all, but returns a negative value for the parameter `ierr`. The default value is 1.

For the ILUT preconditioner:

`ipar[1]` - specifies the destination of error messages generated by the ILUT routine. The default value 6 means that all messages are displayed on the screen. Otherwise routine creates a log file called `MKL_PREC_log.txt` and writes error messages to it. Note if the parameter `ipar[5]` is set to 0, then error messages are not generated at all.

`ipar[5]` - specifies whether error messages are generated. If its value is not equal to 0, the ILUT routine returns error messages as specified by the parameter `ipar[1]`. Otherwise, the routine does not generate error messages at all, but returns a negative value for the parameter `ierr`. The default value is 1.

`ipar[6]` - if its value is greater than 0, the ILUT routine generates warning messages as specified by the parameter `ipar[1]` and continues calculations. If its value is equal to 0, the routine returns a positive value of the parameter `ierr`. If its value is less than 0, the routine generates a warning message as specified by the parameter `ipar[1]` and returns a positive value of the parameter `ierr`. The default value is 1.

dcsrilu0

ILU0 preconditioner based on incomplete LU factorization of a sparse matrix.

Syntax

```
void dcsrilu0 (const MKL_INT *n , const double *a , const MKL_INT *ia , const MKL_INT
             *ja , double *bilu0 , const MKL_INT *ipar , const double *dpar , MKL_INT *ierr );
```

Include Files

- `mkl.h`

Description

The routine `dcsrilu0` computes a preconditioner B [Saad03] of a given sparse matrix A stored in the format accepted in the direct sparse solvers:

$A \sim B = L * U$, where L is a lower triangular matrix with a unit diagonal, U is an upper triangular matrix with a non-unit diagonal, and the portrait of the original matrix A is used to store the incomplete factors L and U .

Caution

This routine supports only one-based indexing of the array parameters.

Input Parameters

<i>n</i>	Size (number of rows or columns) of the original square <i>n</i> -by- <i>n</i> matrix <i>A</i> .
<i>a</i>	Array containing the set of elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to the <i>values</i> array description in the Sparse Matrix Storage Format for more details.
<i>ia</i>	Array of size $(n+1)$ containing begin indices of rows of the matrix <i>A</i> such that <i>ia</i> [<i>i</i>] is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> [<i>n</i>] is equal to the number of non-zero elements in the matrix <i>A</i> , plus one. Refer to the <i>rowIndex</i> array description in the Sparse Matrix Storage Format for more details.
<i>ja</i>	Array containing the column indices for each non-zero element of the matrix <i>A</i> . It is important that the indices are in increasing order per row. The matrix size is equal to the size of the array <i>a</i> . Refer to the <i>columns</i> array description in the Sparse Matrix Storage Format for more details.

Caution

If column indices are not stored in ascending order for each row of matrix, the result of the routine might not be correct.

<i>ipar</i>	Array of size 128. This parameter specifies the integer set of data for both the ILU0 and RCI FGMRES computations. Refer to the <i>ipar</i> array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries that are specific to ILU0 are listed below.
-------------	---

<i>ipar</i> [30]	specifies how the routine operates when a zero diagonal element occurs during calculation. If this parameter is set to 0 (the default value set by the routine dfgmres_init), then the calculations are stopped and the routine returns a non-zero error value. Otherwise, the diagonal element is set to the value of <i>dpar</i> [31] and the calculations continue.
------------------	---

NOTE

You can declare the *ipar* array with a size of 32. However, for future compatibility you must declare the array *ipar* with length 128.

<i>dpar</i>	Array of size 128. This parameter specifies the double precision set of data for both the ILU0 and RCI FGMRES computations. Refer to the <i>dpar</i> array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries specific to ILU0 are listed below.
-------------	---

<i>dpar</i> [30]	specifies a small value, which is compared with the computed diagonal elements. When <i>ipar</i> [30] is not 0, then diagonal elements less than <i>dpar</i> [30] are set to <i>dpar</i> [31]. The default value is 1.0e-16.
------------------	--

NOTE

This parameter can be set to the negative value, because the calculation uses its absolute value.

If this parameter is set to 0, the comparison with the diagonal element is not performed.

dpar[31]

specifies the value that is assigned to the diagonal element if its value is less than *dpar*[30] (see above). The default value is 1.0e-10.

NOTE

You can declare the *dpar* array with a size of 32. However, for future compatibility you must declare the array *dpar* with length 128.

Output Parameters

bilu0

Array *B* containing non-zero elements of the resulting preconditioning matrix *B*, stored in the format accepted in direct sparse solvers. Its size is equal to the number of non-zero elements in the matrix *A*. Refer to the *values* array description in the [Sparse Matrix Storage Format](#) section for more details.

ierr

Error flag, gives information about the routine completion.

NOTE

To present the resulting preconditioning matrix in the CSR3 format the arrays *ia* (row indices) and *ja* (column indices) of the input matrix must be used.

Return Values

ierr=0

Indicates that the task completed normally.

ierr=-101

Indicates that the routine was interrupted and that error occurred: at least one diagonal element is omitted from the matrix in CSR3 format (see [Sparse Matrix Storage Format](#)).

ierr=-102

Indicates that the routine was interrupted because the matrix contains a diagonal element with the value of zero.

ierr=-103

Indicates that the routine was interrupted because the matrix contains a diagonal element which is so small that it could cause an overflow, or that it would cause a bad approximation to ILU0.

ierr=-104

Indicates that the routine was interrupted because the memory is insufficient for the internal work array.

`ierr=-105`

Indicates that the routine was interrupted because the input matrix size n is less than or equal to 0.

`ierr=-106`

Indicates that the routine was interrupted because the column indices ja are not in the ascending order.

dcsrilit

ILUT preconditioner based on the incomplete LU factorization with a threshold of a sparse matrix.

Syntax

```
void dcsrilit (const MKL_INT *n, const double *a, const MKL_INT *ia, const MKL_INT *ja,
double *bilut, MKL_INT *ibilut, MKL_INT *jbilut, const double *tol, const MKL_INT
*maxfil, const MKL_INT *ipar, const double *dpar, MKL_INT *ierr);
```

Include Files

- mkl.h

Description

The routine `dcsrilit` computes a preconditioner B [Saad03] of a given sparse matrix A stored in the format accepted in the direct sparse solvers:

$A \sim B = L * U$, where L is a lower triangular matrix with unit diagonal and U is an upper triangular matrix with non-unit diagonal.

The following threshold criteria are used to generate the incomplete factors L and U :

- 1) the resulting entry must be greater than the matrix current row norm multiplied by the parameter `tol`, and
- 2) the number of the non-zero elements in each row of the resulting L and U factors must not be greater than the value of the parameter `maxfil`.

Caution

This routine supports only one-based indexing of the array parameters.

Input Parameters

n	Size (number of rows or columns) of the original square n -by- n matrix A .
a	Array containing all non-zero elements of the matrix A . The length of the array is equal to their number. Refer to <code>values</code> array description in the Sparse Matrix Storage Format section for more details.
ia	Array of size $(n+1)$ containing indices of non-zero elements in the array a . $ia[i]$ is the index of the first non-zero element from the row i . The value of the last element $ia[n]$ is equal to the number of non-zeros in the matrix A , plus one. Refer to the <code>rowIndex</code> array description in the Sparse Matrix Storage Format for more details.

ja Array of size equal to the size of the array *a*. This array contains the column numbers for each non-zero element of the matrix *A*. It is important that the indices are in increasing order per row. Refer to the *columns* array description in the [Sparse Matrix Storage Format](#) for more details.

Caution

If column indices are not stored in ascending order for each row of matrix, the result of the routine might not be correct.

tol Tolerance for threshold criterion for the resulting entries of the preconditioner.

maxfil Maximum fill-in, which is half of the preconditioner bandwidth. The number of non-zero elements in the rows of the preconditioner cannot exceed $(2*maxfil+1)$.

ipar Array of size 128. This parameter is used to specify the integer set of data for both the ILUT and RCI FGMRES computations. Refer to the *ipar* array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries specific to ILUT are listed below.

ipar[30] specifies how the routine operates if the value of the computed diagonal element is less than the current matrix row norm multiplied by the value of the parameter *tol*. If *ipar*[30] = 0, then the calculation is stopped and the routine returns non-zero error value. Otherwise, the value of the diagonal element is set to a value determined by *dpar*[30] (see its description below), and the calculations continue.

NOTE

There is no default value for *ipar*[30] even if the preconditioner is used within the RCI ISS context. Always set the value of this entry.

NOTE

You must declare the array *ipar* with length 128. While defining the array in the code as *ipar*[30] works, there is no guarantee of future compatibility with Intel® oneAPI Math Kernel Library (oneMKL).

dpar Array of size 128. This parameter specifies the double precision set of data for both ILUT and RCI FGMRES computations. Refer to the *dpar* array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries that are specific to ILUT are listed below.

`dpar[30]`

used to adjust the value of small diagonal elements. Diagonal elements with a value less than the current matrix row norm multiplied by *tol* are replaced with the value of *dpar[30]* multiplied by the matrix row norm.

NOTE

There is no default value for *dpar[30]* entry even if the preconditioner is used within RCI ISS context. Always set the value of this entry.

NOTE

You must declare the array *dpar* with length 128. While defining the array in the code as *ipar[30]* works, there is no guarantee of future compatibility with Intel® oneAPI Math Kernel Library (oneMKL).

Output Parameters

`bilut`

Array containing non-zero elements of the resulting preconditioning matrix *B*, stored in the format accepted in the direct sparse solvers. Refer to the *values* array description in the [Sparse Matrix Storage Format](#) for more details. The size of the array is equal to $(2 * \text{maxfil} + 1) * n - \text{maxfil} * (\text{maxfil} + 1) + 1$.

NOTE

Provide enough memory for this array before calling the routine. Otherwise, the routine may fail to complete successfully with a correct result.

`ibilut`

Array of size $(n+1)$ containing indices of non-zero elements in the array *bilut*. *ibilut[i]* is the index of the first non-zero element from the row *i*. The value of the last element *ibilut[n]* is equal to the number of non-zeros in the matrix *B*, plus one. Refer to the *rowIndex* array description in the [Sparse Matrix Storage Format](#) for more details.

`jbilut`

Array, its size is equal to the size of the array *bilut*. This array contains the column numbers for each non-zero element of the matrix *B*. Refer to the *columns* array description in the [Sparse Matrix Storage Format](#) for more details.

`ierr`

Error flag, gives information about the routine completion.

Return Values

`ierr=0`

Indicates that the task completed normally.

<code>ierr=-101</code>	Indicates that the routine was interrupted because of an error: the number of elements in some matrix row specified in the sparse format is equal to or less than 0.
<code>ierr=-102</code>	Indicates that the routine was interrupted because the value of the computed diagonal element is less than the product of the given tolerance and the current matrix row norm, and it cannot be replaced as <code>ipar[30]=0</code> .
<code>ierr=-103</code>	Indicates that the routine was interrupted because the element <code>ia[i]</code> is less than or equal to the element <code>ia[i - 1]</code> (see Sparse Matrix Storage Format).
<code>ierr=-104</code>	Indicates that the routine was interrupted because the memory is insufficient for the internal work arrays.
<code>ierr=-105</code>	Indicates that the routine was interrupted because the input value of <code>maxfil</code> is less than 0.
<code>ierr=-106</code>	Indicates that the routine was interrupted because the size <code>n</code> of the input matrix is less than 0.
<code>ierr=-107</code>	Indicates that the routine was interrupted because an element of the array <code>ja</code> is less than 1, or greater than <code>n</code> (see Sparse Matrix Storage Format).
<code>ierr=101</code>	The value of <code>maxfil</code> is greater than or equal to <code>n</code> . The calculation is performed with the value of <code>maxfil</code> set to <code>(n-1)</code> .
<code>ierr=102</code>	The value of <code>tol</code> is less than 0. The calculation is performed with the value of the parameter set to <code>(-tol)</code> .
<code>ierr=103</code>	The absolute value of <code>tol</code> is greater than value of <code>dpar[30]</code> ; it can result in instability of the calculation.
<code>ierr=104</code>	The value of <code>dpar[30]</code> is equal to 0. It can cause calculations to fail.

Sparse Matrix Checker Routines

Intel® oneAPI Math Kernel Library (oneMKL) provides a sparse matrix checker so that you can find errors in the storage of sparse matrices before calling Intel® oneAPI Math Kernel Library (oneMKL) PARDISO, DSS, or Sparse BLAS routines.

`sparse_matrix_checker`

Checks the correctness of a sparse matrix.

Syntax

```
MKL_INT sparse_matrix_checker (sparse_struct* handle);
```

Include Files

- `mkl.h`

Description

The `sparse_matrix_checker` routine checks a user-defined array used to store a sparse matrix in order to detect issues which could cause problems in routines that require sparse input matrices, such as Intel® oneAPI Math Kernel Library (oneMKL) PARDISO, DSS, or Sparse BLAS.

Input Parameters

handle Pointer to the data structure describing the sparse array to check.

Return Values

The routine returns a value *error*. Additionally, the *check_result* parameter returns information about where the error occurred, which can be used when *message_level* is `MKL_NO_PRINT`.

Sparse Matrix Checker Error Values

<i>error</i> value	Meaning	Location
<code>MKL_SPARSE_CHECKER_SUCCESS</code>	The input array successfully passed all checks.	
<code>MKL_SPARSE_CHECKER_NON_MONOTONIC</code>	The input array is not 0 or 1 based (<i>ia</i> [0] is not 0 or 1) or elements of <i>ia</i> are not in non-decreasing order as required.	C: <i>ia</i> [<i>i</i>] and <i>ia</i> [<i>i</i> + 1] are incompatible. <i>check_result</i> [0] = <i>i</i> <i>check_result</i> [1] = <i>ia</i> [<i>i</i>] <i>check_result</i> [2] = <i>ia</i> [<i>i</i> + 1]
<code>MKL_SPARSE_CHECKER_OUT_OF_RANGE</code>	The value of the <i>ja</i> array is lower than the number of the first column or greater than the number of the last column.	C: <i>ia</i> [<i>i</i>] and <i>ia</i> [<i>i</i> + 1] are incompatible. <i>check_result</i> [0] = <i>i</i> <i>check_result</i> [1] = <i>ia</i> [<i>i</i>] <i>check_result</i> [2] = <i>ia</i> [<i>i</i> + 1]
<code>MKL_SPARSE_CHECKER_NON_TRIANGULAR</code>	The <i>matrix_structure</i> parameter is <code>MKL_UPPER_TRIANGULAR</code> and both <i>ia</i> and <i>ja</i> are not upper triangular, or the <i>matrix_structure</i> parameter is <code>MKL_LOWER_TRIANGULAR</code> and both <i>ia</i> and <i>ja</i> are not lower triangular	C: <i>ia</i> [<i>i</i>] and <i>ja</i> [<i>j</i>] are incompatible. <i>check_result</i> [0] = <i>i</i> <i>check_result</i> [1] = <i>ia</i> [<i>i</i>] = <i>j</i> <i>check_result</i> [2] = <i>ja</i> [<i>j</i>]
<code>MKL_SPARSE_CHECKER_NON_ORDERED</code>	The elements of the <i>ja</i> array are not in non-decreasing order in each row as required.	C: <i>ia</i> [<i>i</i>] and <i>ia</i> [<i>i</i> + 1] are incompatible. <i>check_result</i> [0] = <i>j</i> <i>check_result</i> [1] = <i>ja</i> [<i>j</i>] <i>check_result</i> [2] = <i>ja</i> [<i>j</i> + 1]

See Also

[sparse_matrix_checker_init](#) Initializes handle for sparse matrix checker.

[Intel® oneAPI Math Kernel Library \(oneMKL\) PARDISO - Parallel Direct Sparse Solver Interface](#)

[Sparse BLAS Level 2 and Level 3 Routines](#)

[Sparse Matrix Storage Formats](#)

sparse_matrix_checker_init

Initializes handle for sparse matrix checker.

Syntax

```
void sparse_matrix_checker_init (sparse_struct* handle);
```

Include Files

- `mkl.h`

Description

The `sparse_matrix_checker_init` routine initializes the handle for the `sparse_matrix_checker` routine. The `handle` variable contains this data:

Description of `sparse_matrix_checker` *handle* Data

Field	Type	Possible Values	Meaning
<code>n</code>	<code>MKL_INT</code>		Order of the matrix stored in sparse array.
<code>csr_ia</code>	<code>MKL_INT*</code>	Pointer to <i>ia</i> array for <code>matrix_format = MKL_CSR</code>	
<code>csr_ja</code>	<code>MKL_INT*</code>	Pointer to <i>ja</i> array for <code>matrix_format = MKL_CSR</code>	
<code>check_result[3]</code>	<code>MKL_INT</code>	See Sparse Matrix Checker Error Values for a description of the values returned in <code>check_result</code> .	Indicates location of problem in array when <code>message_level = MKL_NO_PRINT</code> .
<code>indexing</code>	<code>sparse_matrix_indexing</code>	<code>MKL_ZERO_BASED</code> <code>MKL_ONE_BASED</code>	Indexing style used in array.
<code>matrix_structure</code>	<code>sparse_matrix_structures</code>	<code>MKL_GENERAL_STRUCTURE</code> <code>MKL_UPPER_TRIANGULAR</code> <code>MKL_LOWER_TRIANGULAR</code> <code>MKL_STRUCTURAL_SYMMETRIC</code>	Type of sparse matrix stored in array.

Field	Type	Possible Values	Meaning
matrix_format	sparse_matrix_formats	MKL_CSR	Format of array used for sparse matrix storage.
message_level	sparse_matrix_message_levels	MKL_NO_PRINT MKL_PRINT	Determines whether or not feedback is provided on the screen.
print_style	sparse_matrix_print_styles	MKL_C_STYLE MKL_FORTRAN_STYLE	Determines style of messages when message_level = MKL_PRINT.

Input Parameters

handle Pointer to the data structure describing the sparse array to check.

Output Parameters

handle Pointer to the initialized data structure.

See Also

[sparse_matrix_checker](#) Checks the correctness of a sparse matrix.

[Intel® oneAPI Math Kernel Library \(oneMKL\) PARDISO - Parallel Direct Sparse Solver Interface](#)

[Sparse BLAS Level 2 and Level 3 Routines](#)

[Sparse Matrix Storage Formats](#)

Extended Eigensolver Routines

NOTE

Intel® oneAPI Math Kernel Library (oneMKL) only supports the shared memory programming (SMP) version of the eigensolver.

- [The FEAST Algorithm](#) gives a brief description of the algorithm underlying the Extended Eigensolver.
- [Extended Eigensolver Functionality](#) describes the problems that can and cannot be solved with the Extended Eigensolver and how to get the best results from the routines.
- [Extended Eigensolver Interfaces](#) gives a reference for calling Extended Eigensolver routines.
-

The FEAST Algorithm

The Extended Eigensolver functionality is a set of high-performance numerical routines for solving symmetric standard eigenvalue problems, $Ax = \lambda x$, or generalized symmetric-definite eigenvalue problems, $Ax = \lambda Bx$. It yields all the eigenvalues (λ) and eigenvectors (x) within a given search interval $[\lambda_{\min}, \lambda_{\max}]$. It is based on the FEAST algorithm, an innovative fast and stable numerical algorithm presented in [Polizzi09], which fundamentally differs from the traditional Krylov subspace iteration based techniques (Arnoldi and Lanczos algorithms [Bai00]) or other Davidson-Jacobi techniques [Sleijpen96]. The FEAST algorithm is inspired by the density-matrix representation and contour integration techniques in quantum mechanics.

The FEAST numerical algorithm obtains eigenpair solutions using a numerically efficient contour integration technique. The main computational tasks in the FEAST algorithm consist of solving a few independent linear systems along the contour and solving a reduced eigenvalue problem. Consider a circle centered in the

middle of the search interval $[\lambda_{\min}, \lambda_{\max}]$. The numerical integration over the circle in the current version of FEAST is performed using N_e -point Gauss-Legendre quadrature with x_e the e -th Gauss node associated with the weight ω_e . For example, for the case $N_e = 8$:

$(x_1, \omega_1) = (0.183434642495649, 0.362683783378361),$
 $(x_2, \omega_2) = (-0.183434642495649, 0.362683783378361),$
 $(x_3, \omega_3) = (0.525532409916328, 0.313706645877887),$
 $(x_4, \omega_4) = (-0.525532409916328, 0.313706645877887),$
 $(x_5, \omega_5) = (0.796666477413626, 0.222381034453374),$
 $(x_6, \omega_6) = (-0.796666477413626, 0.222381034453374),$
 $(x_7, \omega_7) = (0.960289856497536, 0.101228536290376),$ and
 $(x_8, \omega_8) = (-0.960289856497536, 0.101228536290376).$

The figure [FEAST Pseudocode](#) shows the basic pseudocode for the FEAST algorithm for the case of real symmetric (left pane) and complex Hermitian (right pane) generalized eigenvalue problems, using N for the size of the system and M for the number of eigenvalues in the search interval (see [\[Polizzi09\]](#)).

NOTE

The pseudocode presents a simplified version of the actual algorithm. Refer to <http://arxiv.org/abs/1302.0432> for an in-depth presentation and mathematical proof of convergence of FEAST.

FEAST Pseudocode

A : real symmetric

B : symmetric positive definite (SPD)

$\Re\{x\}$: real part of x

A : complex Hermitian

B : Hermitian positive definite (HPD)

- | | |
|--|--|
| <ol style="list-style-type: none"> 1. Select $M_0 > M$ random vectors $Y_{N \times M_0}$. 2. Set $Q = 0$ with $Q \in \mathbb{R}^{N \times M_0}$; $r = (\lambda_{\max} - \lambda_{\min}) / 2$;
For $e = 1, \dots, N_e$
compute $\theta_e = -(\pi / 2)(x_e - 1)$,
compute $Z_e = (\lambda_{\max} + \lambda_{\min}) / 2 + r \exp(i\theta_e)$,
solve $(Z_e B - A)Q_e = Y$ to obtain $Q_e \in \mathbb{R}^{N \times M_0}$
compute $Q = Q - (\omega_e / 2) \Re\{r \exp(i\theta_e) Q_e\}$
End 3. Form $A_{Q_{M_0 \times M_0}} = Q^T A Q$ and $B_{Q_{M_0 \times M_0}} = Q^T B Q$
reduce value of M_0 if B_Q is not symmetric positive definite. 4. Solve $A_Q \Phi = \varepsilon B_Q \Phi$ to obtain the M_0 eigenvalue ε_m, and eigenvectors $\Phi_{M_0 \times M_0} \in \mathbb{R}^{M_0 \times M_0}$. 5. Set $\lambda_m = \varepsilon_m$ and compute $X_{N \times M_0} = Q_{N \times M_0} \Phi_{M_0 \times M_0}$.
If $\lambda_m \in [\lambda_{\min}, \lambda_{\max}]$, λ_m is an eigenvalue and its eigenvector is X_m (the m-th column of X). 6. Check convergence for the trace of eigenvalues λ_m. If iterative refinement needed, compute $Y = BX$ and go back to step 2. | <ol style="list-style-type: none"> 1. Select $M_0 > M$ random vectors $Y_{N \times M_0} \in \mathbb{C}^{N \times M_0}$. 2. Set $Q = 0$ with $Q \in \mathbb{R}^{N \times M_0}$; $r = (\lambda_{\max} - \lambda_{\min}) / 2$;
For $e = 1, \dots, N_e$
compute $\theta_e = -(\pi / 2)(x_e - 1)$,
compute $Z_e = (\lambda_{\max} + \lambda_{\min}) / 2 + r \exp(i\theta_e)$,
solve $(Z_e B - A)Q_e = Y$ to obtain $Q_e \in \mathbb{C}^{N \times M_0}$
solve $(Z_e B - A)^H \hat{Q}_e = Y$ to obtain $\hat{Q}_e \in \mathbb{C}^{N \times M_0}$
$Q = Q - (\omega_e / 4) r (\exp(i\theta_e) Q_e + \exp(-i\theta_e) \hat{Q}_e)$
End 3. Form $A_{Q_{M_0 \times M_0}} = Q^H A Q$ and $B_{Q_{M_0 \times M_0}} = Q^H B Q$
reduce value of M_0 if B_Q is not Hermitian positive definite. 4. Solve $A_Q \Phi = \varepsilon B_Q \Phi$ to obtain the M_0 eigenvalue ε_m, and eigenvectors $\Phi_{M_0 \times M_0} \in \mathbb{C}^{M_0 \times M_0}$. 5. Set $\lambda_m = \varepsilon_m$ and compute $X_{N \times M_0} = Q_{N \times M_0} \Phi_{M_0 \times M_0}$.
If $\lambda_m \in [\lambda_{\min}, \lambda_{\max}]$, λ_m is an eigenvalue solution and its eigenvector is X_m (the m-th column of X). 6. Check convergence for the trace of the eigenvalues λ_m. If iterative refinement is needed, compute $Y = BX$ and go back to step 2. |
|--|--|

Extended Eigensolver Functionality

The eigenvalue problems covered are as follows:

- standard, $Ax = \lambda x$
 - A complex Hermitian
 - A real symmetric
- generalized, $Ax = \lambda Bx$
 - A complex Hermitian, B Hermitian positive definite (hpd)
 - A real symmetric and B real symmetric positive definite (spd)

The Extended Eigensolver functionality offers:

- Real/Complex and Single/Double precisions: double precision is recommended to provide better accuracy of eigenpairs.
- Reverse communication interfaces (RCI) provide maximum flexibility for specific applications. RCI are independent of matrix format and inner system solvers, so you must provide your own linear system solvers (direct or iterative) and matrix-matrix multiply routines.
- Predefined driver interfaces for dense, LAPACK banded, and sparse (CSR) formats are less flexible but are optimized and easy to use:
 - The Extended Eigensolver interfaces for dense matrices are likely to be slower than the comparable LAPACK routines because the FEAST algorithm has a higher computational cost.
 - The Extended Eigensolver interfaces for banded matrices support banded LAPACK-type storage.
 - The Extended Eigensolver sparse interfaces support compressed sparse row format and use the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver.

Product and Performance Information

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

Notice revision #20201201

Parallelism in Extended Eigensolver Routines

How you achieve parallelism in Extended Eigensolver routines depends on which interface you use. Parallelism (via shared memory programming) is not *explicitly* implemented in Extended Eigensolver routines within one node: the inner linear systems are currently solved one after another.

- Using the Extended Eigensolver RCI interfaces, you can achieve parallelism by providing a threaded inner system solver and a matrix-matrix multiplication routine. When using the RCI interfaces, you are responsible for activating the threaded capabilities of your BLAS and LAPACK libraries most likely using the shell variable `OMP_NUM_THREADS`.
- Using the predefined Extended Eigensolver interfaces, parallelism can be implicitly obtained within the shared memory version of BLAS, LAPACK or Intel® oneAPI Math Kernel Library (oneMKL) PARDISO. The shell variable `MKL_NUM_THREADS` can be used for automatically setting the number of OpenMP threads (cores) for BLAS, LAPACK, and Intel® oneAPI Math Kernel Library (oneMKL) PARDISO.

Product and Performance Information

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

Notice revision #20201201

Achieving Performance With Extended Eigensolver Routines

In order to use the Extended Eigensolver Routines, you need to provide

- the search interval and the size of the subspace M_0 (overestimation of the number of eigenvalues M within a given search interval);
- the system matrix in dense, banded, or sparse CSR format if the Extended Eigensolver predefined interfaces are used, or a high-performance complex direct or iterative system solver and matrix-vector multiplication routine if RCI interfaces are used.

In return, you can expect

- fast convergence with very high accuracy when seeking up to 1000 eigenpairs (in two to four iterations using $M_0 = 1.5M$, and $N_e = 8$ or at most using $N_e = 16$ contour points);
- an extremely robust approach.

The performance of the basic FEAST algorithm depends on a trade-off between the choices of the number of Gauss quadrature points N_e , the size of the subspace M_0 , and the number of outer refinement loops to reach the desired accuracy. In practice you should use $M_0 > 1.5 M$, $N_e = 8$, and at most two refinement loops.

For better performance:

- M_0 should be much smaller than the size of the eigenvalue problem, so that the arithmetic complexity depends mainly on the inner system solver ($O(NM)$ for narrow-banded or sparse systems).
- Parallel scalability performance depends on the shared memory capabilities of the of the inner system solver.
- For very large sparse and challenging systems, application users should make use of the Extended Eigensolver RCI interfaces with customized highly-efficient iterative systems solvers and preconditioners.
- For the Extended Eigensolver interfaces for banded matrices, the parallel performance scalability is limited.

Product and Performance Information

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

Notice revision #20201201

Extended Eigensolver Interfaces for Eigenvalues within Interval

Extended Eigensolver Naming Conventions

There are two different types of interfaces available in the Extended Eigensolver routines:

1. The reverse communication interfaces (RCI):

```
?feast_<matrix type>_rci
```

These interfaces are matrix free format (the interfaces are independent of the matrix data formats). You must provide matrix-vector multiply and direct/iterative linear system solvers for your own explicit or implicit data format.

2. The predefined interfaces:

```
?feast_<matrix type><type of eigenvalue problem>
```

are predefined drivers for `?feast` reverse communication interface that act on commonly used matrix data storage (dense, banded and compressed sparse row representation), using internal matrix-vector routines and selected inner linear system solvers.

For these interfaces:

- ? indicates the data type of matrix *A* (and matrix *B* if any) defined as follows:

s	float
d	double
c	MKL_Complex8
z	MKL_Complex16

- <matrix type> defined as follows:

Value of <matrix type>	Matrix format	Inner linear system solver used by Extended Eigensolver
sy (symmetric real)		
he (Hermitian complex)	Dense	LAPACK dense solvers
sb (symmetric banded real)	Banded-LAPACK	Internal banded solver

Value of <i><matrix type></i>	Matrix format	Inner linear system solver used by Extended Eigensolver
hb	(Hermitian banded complex)	PARDISO solver
scsr	(symmetric real)	
hcsr	(Hermitian complex)	
s	(symmetric real)	User defined
h	(Hermitian complex)	

- <type of eigenvalue problem>* is:

gv	generalized eigenvalue problem
ev	standard eigenvalue problem

For example, `sfeast_scsrev` is a single-precision routine with a symmetric real matrix stored in sparse compressed-row format for a standard eigenvalue problem, and `zfeast_hrci` is a complex double-precision routine with a Hermitian matrix using the reverse communication interface.

Note that:

- ? can be `s` or `d` if a matrix is real symmetric: *<matrix type>* is `sy`, `sb`, or `scsr`.
- ? can be `c` or `z` if a matrix is complex Hermitian: *<matrix type>* is `he`, `hb`, or `hcsr`.
- ? can be `c` or `z` if the Extended Eigensolver RCI interface is used for solving a complex Hermitian problem.
- ? can be `s` or `d` if the Extended Eigensolver RCI interface is used for solving a real symmetric problem.

feastinit

Initialize Extended Eigensolver input parameters with default values.

Syntax

```
feastinit (MKL_INT* fpm);
```

Include Files

- `mkl.h`

Description

This routine sets all Extended Eigensolver parameters to their default values.

Output Parameters

<i>fpm</i>	Array, size 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
------------	---

Extended Eigensolver Input Parameters

The input parameters for Extended Eigensolver routines are contained in an `MKL_INT` array named *fpm*. To call the Extended Eigensolver interfaces, this array should be initialized using the routine `feastinit`.

Parameter	Default	Description
$fpm[0]$	0	Specifies whether Extended Eigensolver routines print runtime status. $fpm[0]=0$ Extended Eigensolver routines do not generate runtime messages at all. $fpm[0]=1$ Extended Eigensolver routines print runtime status to the screen.
$fpm[1]$	8	The number of contour points $N_e = 8$ (see the description of FEAST algorithm). Must be one of $\{3,4,5,6,8,10,12,16,20,24,32,40,48\}$.
$fpm[2]$	12	Error trace double precision stopping criteria ε ($\varepsilon = 10^{-fpm[2]}$).
$fpm[3]$	20	Maximum number of Extended Eigensolver refinement loops allowed. If no convergence is reached within $fpm[3]$ refinement loops, Extended Eigensolver routines return $info=2$.
$fpm[4]$	0	User initial subspace. If $fpm[4]=0$ then Extended Eigensolver routines generate initial subspace, if $fpm[4]=1$ the user supplied initial subspace is used.
$fpm[5]$	0	Extended Eigensolver stopping test. $fpm[5]=0$ Extended Eigensolvers are stopped if this residual stopping test is satisfied: <ul style="list-style-type: none"> generalized eigenvalue problem: $\max_{i=1:mode} \frac{\ Ax_i - \lambda_i Bx_i\ _1}{\max(E_{\min} , E_{\max}) \ Bx_i\ _1} < \varepsilon$ standard eigenvalue problem: $\max_{i=1:mode} \frac{\ Ax_i - \lambda_i x_i\ _1}{\max(E_{\min} , E_{\max}) \ x_i\ _1} < \varepsilon$ where $mode$ is the total number of eigenvalues found in the search interval and $\varepsilon = 10^{-fpm[6]}$ for real and complex or $\varepsilon = 10^{-fpm[2]}$ for double precision and double complex. $fpm[5]=1$ Extended Eigensolvers are stopped if this trace stopping test is satisfied: $\frac{ trace_j - trace_{j-1} }{\max(E_{\min} , E_{\max})} < \varepsilon$, where $trace_j$ denotes the sum of all eigenvalues found in the search interval $[emin, emax]$ at the j -th Extended Eigensolver iteration: $trace_j = \sum_{i=1}^M \lambda_i^{(j)}$.
$fpm[6]$	5	Error trace single precision stopping criteria ($10^{-fpm[6]}$).
$fpm[13]$	0	$fpm[13]=0$ Standard use for Extended Eigensolver routines.

Parameter	Default	Description
		$fpm[13]=1$ Non-standard use for Extended Eigensolver routines: return the computed eigenvectors subspace after one single contour integration.
$fpm[26]$	0	Specifies whether Extended Eigensolver routines check input matrices (applies to CSR format only).
		$fpm[26]=0$ Extended Eigensolver routines do not check input matrices.
		$fpm[26]=1$ Extended Eigensolver routines check input matrices.
$fpm[27]$	0	Check if matrix B is positive definite. Set $fpm[27] = 1$ to check if B is positive definite.
$fpm[29]$ to $fpm[62]$	-	Reserved for future use.
$fpm[63]$	0	Use the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO solver with the user-defined PARDISO $iparm$ array settings.
NOTE This option can only be used by Extended Eigensolver Predefined Interfaces for Sparse Matrices.		
		$fpm[63]=0$ Extended Eigensolver routines use the Intel® oneAPI Math Kernel Library (oneMKL) PARDISO default $iparm$ settings defined by calling the <code>pardisoinit</code> subroutine.
		$fpm[63]=1$ The values from $fpm[64]$ to $fpm[127]$ correspond to $iparm[0]$ to $iparm[63]$ respectively according to the formula $fpm[64 + i] = iparm[i]$ for $i = 0, 1, \dots, 63$.

Extended Eigensolver Output Details

Errors and warnings encountered during a run of the Extended Eigensolver routines are stored in an integer variable, *info*. If the value of the output *info* parameter is not 0, either an error or warning was encountered. The possible return values for the *info* parameter along with the error code descriptions are given in the following table.

Return Codes for info Parameter

<i>info</i>	Classification	Description
202	Error	Problem with size of the system n ($n \leq 0$)
201	Error	Problem with size of initial subspace $m0$ ($m0 \leq 0$ or $m0 > n$)
200	Error	Problem with $emin, emax$ ($emin \geq emax$)
(100+i)	Error	Problem with i -th value of the input Extended Eigensolver parameter ($fpm[i - 1]$). Only the parameters in use are checked.
4	Warning	Successful return of only the computed subspace after call with $fpm[13] = 1$

<i>info</i>	Classification	Description
3	Warning	Size of the subspace $m0$ is too small ($m0 < m$)
2	Warning	No Convergence (number of iteration loops $> fpm[3]$)
1	Warning	No eigenvalue found in the search interval. See remark below for further details.
0	Successful exit	
-1	Error	Internal error for allocation memory.
-2	Error	Internal error of the inner system solver. Possible reasons: not enough memory for inner linear system solver or inconsistent input.
-3	Error	Internal error of the reduced eigenvalue solver Possible cause: matrix B may not be positive definite. It can be checked by setting $fpm[27] = 1$ before calling an Extended Eigensolver routine, or by using LAPACK routines.
-4	Error	Matrix B is not positive definite.
-(100+i)	Error	Problem with the i -th argument of the Extended Eigensolver interface.

In some extreme cases the return value `info=1` may indicate that the Extended Eigensolver routine has failed to find the eigenvalues in the search interval. This situation could arise if a very large search interval is used to locate a small and isolated cluster of eigenvalues (i.e. the dimension of the search interval is many orders of magnitude larger than the number of contour points. It is then either recommended to increase the number of contour points `fpm[1]` or simply rescale more appropriately the search interval. Rescaling means the initial problem of finding all eigenvalues the search interval $[\lambda_{\min}, \lambda_{\max}]$ for the standard eigenvalue problem $Ax = \lambda x$ is replaced with the problem of finding all eigenvalues in the search interval $[\lambda_{\min}/t, \lambda_{\max}/t]$ for the standard eigenvalue problem $(A/t)x = (\lambda/t)x$ where t is a scaling factor.

Extended Eigensolver RCI Routines

If you do not require specific linear system solvers or matrix storage schemes, you can skip this section and go directly to [Extended Eigensolver Predefined Interfaces](#).

Extended Eigensolver RCI Interface Description

The Extended Eigensolver RCI interfaces can be used to solve standard or generalized eigenvalue problems, and are independent of the format of the matrices. As mentioned earlier, the Extended Eigensolver algorithm is based on the contour integration techniques of the matrix resolvent $G(\sigma) = (\sigma B - A)^{-1}$ over a circle. For solving a generalized eigenvalue problem, Extended Eigensolver has to perform one or more of the following operations at each contour point denoted below by Z_e :

- Factorize the matrix $(Z_e * B - A)$
- Solve the linear system $(Z_e * B - A)X = Y$ or $(Z_e * B - A)^H X = Y$ with multiple right hand sides, where H means transpose conjugate
- Matrix-matrix multiply $BX = Y$ or $AX = Y$

For solving a standard eigenvalue problem, replace the matrix B with the identity matrix I .

The primary aim of RCI interfaces is to isolate these operations: the linear system solver, factorization of the matrix resolvent at each contour point, and matrix-matrix multiplication. This gives universality to RCI interfaces as they are independent of data structures and the specific implementation of the operations like matrix-vector multiplication or inner system solvers. However, this approach requires some additional effort when calling the interface. In particular, operations listed above are performed by routines that you supply on data structures that you find most appropriate for the problem at hand.

To initialize an Extended Eigensolver RCI routine, set the job indicator (*ijob*) parameter to the value -1. When the routine requires the results of an operation, it generates a special value of *ijob* to indicate the operation that needs to be performed. The routine also returns *ze*, the coordinate along the complex contour, the values of array *work* or *workc*, and the number of columns to be used. Your subroutine then must perform the operation at the given contour point *ze*, store the results in prescribed array, and return control to the Extended Eigensolver RCI routine.

The following pseudocode shows the general scheme for using the Extended Eigensolver RCI functionality for a real symmetric problem:

```

ijob=-1; // initialization
do while (ijob!=0) {
    ?feast_src1(&ijob, &N, &Ze, work, workc, Aq, Bq,
               fpm, &epsout, &loop, &Emin, &Emax, &M0, E, lambda, &q, res, &info);

    switch(ijob) {
    case 10: // Factorize the complex matrix (ZeB-A)
        break;

    case 11: // Solve the complex linear system (ZeB-A)x=workc
        // Put result in workc
        break;

    case 30: // Perform multiplication A by Qi..Qj columns of QNxM0
        // where i = fpm[23] and j = fpm[23]+fpm[24]-1
        // Qi..Qj located in q starting from q+N*(i-1)
        break;

    case 40: // Perform multiplication B by Qi..Qj columns of QNxM0
        // where i = fpm[23] and j = fpm[23]+fpm[24]-1
        // Qi..Qj located in q starting from q+N*(i-1)
        // Result is stored in work+N*(i-1)

        break;

    }
}

```

NOTE

The ? option in ?feast in the pseudocode given above should be replaced by either s or d, depending on the matrix data type of the eigenvalue system.

The next pseudocode shows the general scheme for using the Extended Eigensolver RCI functionality for a complex Hermitian problem:

```

ijob=-1; // initialization
while (ijob!=0) {
    ?feast_hrc1(&ijob, &N, &Ze, work, workc, Aq, Bq,
               fpm, &epsout, &loop, &Emin, &Emax, &M0, E, lambda, &q, res, &info);

    switch (ijob) {
    case 10: // Factorize the complex matrix (ZeB-A)
        break;

    case 11: // Solve the linear system (ZeB-A)y=workc
        // Put result in workc
        break;

    case 20: // Factorize (if needed by case 21) the complex matrix (ZeB-A)H

```

```

        // ATTENTION: This option requires additional memory storage
        // (i.e . the resulting matrix from case 10 cannot be overwritten)
break;

case 21: // Solve the linear system (ZeB-A)^Hy=workc
        // Put result in workc
        // REMARK: case 20 becomes obsolete if this solve can be performed
        // using the factorization in case 10
break;

case 30: // Multiply A by Qi..Qj columns of QNxM0,
        // where i = fpm[23] and j = fpm[23]+fpm[24]-1
        // Qi..Qj located in q starting from q+N*(i-1)
        // Result is stored in work+N*(i-1)
break;

case 40: // Perform multiplication B by Qi..Qj columns of QNxM0
        // where i = fpm[23] and j = fpm[23]+fpm[24]-1
        // Qi..Qj located in q starting from q+N*(i-1)
        // Result is stored in work+N*(i-1)
break;

    }
}
end do

```

NOTE

The ? option in ?feast in the pseudocode given above should be replaced by either c or z, depending on the matrix data type of the eigenvalue system.

If case 20 can be avoided, performance could be up to twice as fast, and Extended Eigensolver functionality would use half of the memory.

If an iterative solver is used along with a preconditioner, the factorization of the preconditioner could be performed with *ijob* = 10 (and *ijob* = 20 if applicable) for a given value of *Ze*, and the associated iterative solve would then be performed with *ijob* = 11 (and *ijob* = 21 if applicable).

Product and Performance Information

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

Notice revision #20201201

?feast_src/?feast_hrci

Extended Eigensolver RCI interface.

Syntax

```

void sfeast_src (MKL_INT* ijob, const MKL_INT* n, MKL_Complex8* ze, float* work,
MKL_Complex8* workc, float* aq, float* sq, MKL_INT* fpm, float* epsout, MKL_INT* loop,
const float* emin, const float* emax, MKL_INT* m0, float* lambda, float* q, MKL_INT* m,
float* res, MKL_INT* info);

```

```

void dfeast_src1 (MKL_INT* ijob, const MKL_INT* n, MKL_Complex16* ze, double* work,
MKL_Complex16* workc, double* aq, double* sq, MKL_INT* fpm, double* epsout, MKL_INT*
loop, const double* emin, const double* emax, MKL_INT* m0, double* lambda, double* q,
MKL_INT* m, double* res, MKL_INT* info);

void cfeast_hrci (MKL_INT* ijob, const MKL_INT* n, MKL_Complex8* ze, MKL_Complex8*
work, MKL_Complex8* workc, MKL_Complex8* aq, MKL_Complex8* sq, MKL_INT* fpm, float*
epsout, MKL_INT* loop, const float* emin, const float* emax, MKL_INT* m0, float* lambda,
MKL_Complex8* q, MKL_INT* m, float* res, MKL_INT* info);

void zfeast_hrci (MKL_INT* ijob, const MKL_INT* n, MKL_Complex16* ze, MKL_Complex16*
work, MKL_Complex16* workc, MKL_Complex16* aq, MKL_Complex16* sq, MKL_INT* fpm, double*
epsout, MKL_INT* loop, const double* emin, const double* emax, MKL_INT* m0, double*
lambda, MKL_Complex16* q, MKL_INT* m, double* res, MKL_INT* info);

```

Include Files

- mkl.h

Description

Compute eigenvalues as described in [Extended Eigensolver RCI Interface Description](#).

Input Parameters

<i>ijob</i>	Job indicator variable. On entry, a call to ?feast_src1/?feast_hrci with <i>ijob</i> =-1 initializes the eigensolver.
<i>n</i>	Sets the size of the problem. $n > 0$.
<i>work</i>	Workspace array of size n by $m0$.
<i>workc</i>	Workspace array of size n by $m0$.
<i>aq, sq</i>	Workspace arrays of size $m0$ by $m0$.
<i>fpm</i>	Array, size of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin, emax</i>	The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

<i>m0</i>	On entry, specifies the initial guess for subspace size to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return <i>info</i> =3.
<i>q</i>	On entry, if <i>fpm</i> [4]=1, the array <i>q</i> of size n by m contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

ijob

On exit, the parameter carries the status flag that indicates the condition of the return. The status information is divided into three categories:

1. A zero value indicates successful completion of the task.
2. A positive value indicates that the solver requires a matrix-vector multiplication or solving a specific system with a complex coefficient.
3. A negative value indicates successful initiation.

A non-zero value of *ijob* specifically means the following:

- *ijob* = 10 - factorize the complex matrix $Z_e * B - A$ at a given contour point Z_e and return the control to the ?feast_srcil/?feast_hrcil routine where Z_e is a complex number meaning contour point and its value is defined internally in ?feast_srcil/?feast_hrcil.
- *ijob* = 11 - solve the complex linear system $(Z_e * B - A) * y = workc$, put the solution in *workc* and return the control to the ?feast_srcil/?feast_hrcil routine.
- *ijob* = 20 - factorize the complex matrix $(Z_e * B - A)^H$ at a given contour point Z_e and return the control to the ?feast_srcil/?feast_hrcil routine where Z_e is a complex number meaning contour point and its value is defined internally in ?feast_srcil/?feast_hrcil.

The symbol X^H means transpose conjugate of matrix X .

- *ijob* = 21 - solve the complex linear system $(Z_e * B - A)^H * y = workc$, put the solution in *workc* and return the control to the ?feast_srcil/?feast_hrcil routine. The case *ijob*=20 becomes obsolete if the solve can be performed using the factorization computed for *ijob*=10.

The symbol X^H mean transpose conjugate of matrix X .

- *ijob* = 30 - multiply matrix A by $Q_j..Q_i$, put the result in $work + N*(i - 1)$, and return the control to the ?feast_srcil/?feast_hrcil routine.
i is *fpm*[24], and *j* is *fpm*[23] + *fpm*[24] - 1.
- *ijob* = 40 - multiply matrix B by $Q_j..Q_i$, put the result in $work + N*(i - 1)$ and return the control to the ?feast_srcil/?feast_hrcil routine. If a standard eigenvalue problem is solved, just return *work* = *q*.
i is *fpm*[24], and *j* is *fpm*[23] + *fpm*[24] - 1.
- *ijob* = -2 - rerun the ?feast_srcil/?feast_hrcil task with the same parameters.

ze

Defines the coordinate along the complex contour. All values of *ze* are generated by ?feast_srcil/?feast_hrcil internally.

fpm

On output, contains coordinates of columns of work array needed for iterative refinement. (See [Extended Eigensolver RCI Interface Description](#).)

epsout

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|emin|, |emax|)$

loop

On output, contains the number of refinement loop executed. Ignored on input.

lambda

Array of length *m0*. On output, the first *m* entries of *lambda* are eigenvalues found in the interval.

<i>q</i>	On output, <i>q</i> contains all eigenvectors corresponding to <i>lambda</i> .
<i>m</i>	The total number of eigenvalues found in the interval [<i>emin</i> , <i>emax</i>]: $0 \leq m \leq m0$.
<i>res</i>	<p>Array of length <i>m0</i>. On exit, the first <i>m</i> components contain the relative residual vector:</p> <ul style="list-style-type: none"> generalized eigenvalue problem: $\frac{\ Ax_i - \lambda_i Bx_i\ _1}{\max(E_{\min} , E_{\max}) \ Bx_i\ _1}$ standard eigenvalue problem: $\frac{\ Ax_i - \lambda_i x_i\ _1}{\max(E_{\min} , E_{\max}) \ x_i\ _1}$ <p>for $i=0, 1, \dots, m-1$, and where <i>m</i> is the total number of eigenvalues found in the search interval.</p>
<i>info</i>	If <i>info</i> =0, the execution is successful. If <i>info</i> \neq 0, see Output Eigensolver info Details .

Extended Eigensolver Predefined Interfaces

The predefined interfaces include routines for standard and generalized eigenvalue problems, and for dense, banded, and sparse matrices.

Matrix Type	Standard Eigenvalue Problem	Generalized Eigenvalue Problem
Dense	?feast_syev ?feast_heev	?feast_sygv ?feast_hgv
Banded	?feast_sbev ?feast_hbev	?feast_sbgv ?feast_hbgv
Sparse	?feast_scsrev ?feast_hcsrev	?feast_scsrgv ?feast_hcsrgv

Matrix Storage

The symmetric and Hermitian matrices used in Extended Eigensolvers predefined interfaces can be stored in full, band, and sparse formats.

- In the full storage format (described in [Full Storage](#) in additional detail) you store all elements, all of the elements in the upper triangle of the matrix, or all of the elements in the lower triangle of the matrix.
- In the band storage format (described in [Band storage](#) in additional detail), you store only the elements along a diagonal band of the matrix.
- In the sparse format (described in [Storage Arrays for a Matrix in CSR Format \(3-Array Variation\)](#)), you store only the non-zero elements of the matrix.

In generalized eigenvalue systems you must use the same family of storage format for both matrices *A* and *B*. The bandwidth can be different for the banded format (*k1b* can be different from *k1a*), and the position of the non-zero elements can also be different for the sparse format (CSR coordinates *ib* and *jb* can be different from *ia* and *ja*).

?feast_syev/?feast_heev

Extended Eigensolver interface for standard eigenvalue problem with dense matrices.

Syntax

```
void sfeast_syev (const char * uplo, const MKL_INT * n, const float * a, const MKL_INT * lda, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);

void dfeast_syev (const char * uplo, const MKL_INT * n, const double * a, const MKL_INT * lda, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double * emax, MKL_INT * m0, double * e, double * x, MKL_INT * m, double * res, MKL_INT * info);

void cfeast_heev (const char * uplo, const MKL_INT * n, const MKL_Complex8 * a, const MKL_INT * lda, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT * m, float * res, MKL_INT * info);

void zfeast_heev (const char * uplo, const MKL_INT * n, const MKL_Complex16 * a, const MKL_INT * lda, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x, MKL_INT * m, double * res, MKL_INT * info);
```

Include Files

- mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

<i>uplo</i>	Must be 'U' or 'L' or 'F' . If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular parts of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular parts of <i>A</i> . If <i>uplo</i> = 'F', <i>a</i> stores the full matrix <i>A</i> .
<i>n</i>	Sets the size of the problem. $n > 0$.
<i>a</i>	Array of dimension <i>lda</i> by <i>n</i> , contains either full matrix <i>A</i> or upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i>
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>fpm</i>	Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin</i> , <i>emax</i>	The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

m0 On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return *info*=3.

x On entry, if *fpm*[4]=1, the array *x* of size n by m contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

epsout On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|emin|, |emax|)$

loop On output, contains the number of refinement loop executed. Ignored on input.

e Array of length *m0*. On output, the first m entries of *e* are eigenvalues found in the interval.

x On output, the first m columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the i -th column of *x* holding the eigenvector associated with *e*[i].

m The total number of eigenvalues found in the interval $[emin, emax]$: $0 \leq m \leq m0$.

res Array of length *m0*. On exit, the first m components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i x_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|x_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info If *info*=0, the execution is successful. If *info* \neq 0, see [Output Eigensolver info Details](#).

?feast_sygv/?feast_hgv

Extended Eigensolver interface for generalized eigenvalue problem with dense matrices.

Syntax

```
void sfeast_sylv (const char * uplo, const MKL_INT * n, const float * a, const MKL_INT
* lda, const float * b, const MKL_INT * ldb, MKL_INT * fpm, float * epsout, MKL_INT *
loop, const float * emin, const float * emax, MKL_INT * m0, float * e, float * x,
MKL_INT * m, float * res, MKL_INT * info);

void dfeast_sylv (const char * uplo, const MKL_INT * n, const double * a, const MKL_INT
* lda, const double * b, const MKL_INT * ldb, MKL_INT * fpm, double * epsout, MKL_INT *
loop, const double * emin, const double * emax, MKL_INT * m0, double * e, double * x,
MKL_INT * m, double * res, MKL_INT * info);

void cfeast_hsgv (const char * uplo, const MKL_INT * n, const MKL_Complex8 * a, const
MKL_INT * lda, const MKL_Complex8 * b, const MKL_INT * ldb, MKL_INT * fpm, float *
epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e,
MKL_Complex8 * x, MKL_INT * m, float * res, MKL_INT * info);

void zfeast_hsgv (const char * uplo, const MKL_INT * n, const MKL_Complex16 * a, const
MKL_INT * lda, const MKL_Complex16 * b, const MKL_INT * ldb, MKL_INT * fpm, double *
epsout, MKL_INT * loop, const double * emin, const double * emax, MKL_INT * m0, double
* e, MKL_Complex16 * x, MKL_INT * m, double * res, MKL_INT * info);
```

Include Files

- mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

Input Parameters

<i>uplo</i>	Must be 'U' or 'L' or 'F' . If <i>UPLO</i> = 'U', <i>a</i> and <i>b</i> store the upper triangular parts of <i>A</i> and <i>B</i> respectively. If <i>UPLO</i> = 'L', <i>a</i> and <i>b</i> store the lower triangular parts of <i>A</i> and <i>B</i> respectively. If <i>UPLO</i> = 'F', <i>a</i> and <i>b</i> store the full matrices <i>A</i> and <i>B</i> respectively.
<i>n</i>	Sets the size of the problem. $n > 0$.
<i>a</i>	Array of dimension <i>lda</i> by <i>n</i> , contains either full matrix <i>A</i> or upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i>
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>b</i>	Array of dimension <i>ldb</i> by <i>n</i> , contains either full matrix <i>B</i> or upper or lower triangular part of the matrix <i>B</i> , as specified by <i>uplo</i>
<i>ldb</i>	The leading dimension of the array <i>B</i> . Must be at least $\max(1, n)$.
<i>fpm</i>	Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin</i> , <i>emax</i>	The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

m0 On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return *info*=3.

x On entry, if *fpm*[4]=1, the array *x* of size n by m contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

epsout On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|emin|, |emax|)$

loop On output, contains the number of refinement loop executed. Ignored on input.

e Array of length *m0*. On output, the first m entries of *e* are eigenvalues found in the interval.

x On output, the first m columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the i -th column of *x* holding the eigenvector associated with *e*[i].

m The total number of eigenvalues found in the interval $[emin, emax]$: $0 \leq m \leq m0$.

res Array of length *m0*. On exit, the first m components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|Bx_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info If *info*=0, the execution is successful. If *info* \neq 0, see [Output Eigensolver info Details](#).

?feast_sbev/?feast_hbev

Extended Eigensolver interface for standard eigenvalue problem with banded matrices.

Syntax

```
void sfeast_sbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
float * a, const MKL_INT * lda, MKL_INT * fpm, float * epsout, MKL_INT * loop, const
float * emin, const float * emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float
* res, MKL_INT * info);
```

```
void dfeast_sbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
double * a, const MKL_INT * lda, MKL_INT * fpm, double * epsout, MKL_INT * loop, const
double * emin, const double * emax, MKL_INT * m0, double * e, double * x, MKL_INT * m,
double * res, MKL_INT * info);
```

```
void cfeast_hbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
MKL_Complex8 * a, const MKL_INT * lda, MKL_INT * fpm, float * epsout, MKL_INT * loop,
const float * emin, const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x,
MKL_INT * m, float * res, MKL_INT * info);
```

```
void zfeast_hbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
MKL_Complex16 * a, const MKL_INT * lda, MKL_INT * fpm, double * epsout, MKL_INT * loop,
const double * emin, const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x,
MKL_INT * m, double * res, MKL_INT * info);
```

Include Files

- mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

<i>uplo</i>	Must be 'U' or 'L' or 'F' . If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular parts of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular parts of <i>A</i> . If <i>uplo</i> = 'F', <i>a</i> stores the full matrix <i>A</i> .
<i>n</i>	Sets the size of the problem. $n > 0$.
<i>kla</i>	The number of super- or sub-diagonals within the band in <i>A</i> ($kla \geq 0$).
<i>a</i>	Array of dimension <i>lda</i> by <i>n</i> , contains either full matrix <i>A</i> or upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i>
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>fpm</i>	Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin</i> , <i>emax</i>	The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

m0 On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return *info*=3.

x On entry, if *fpm*[4]=1, the array *x* of size n by m contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

epsout On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|emin|, |emax|)$

loop On output, contains the number of refinement loop executed. Ignored on input.

e Array of length *m0*. On output, the first m entries of *e* are eigenvalues found in the interval.

x On output, the first m columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the i -th column of *x* holding the eigenvector associated with *e*[i].

m The total number of eigenvalues found in the interval $[emin, emax]$: $0 \leq m \leq m0$.

res Array of length *m0*. On exit, the first m components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i x_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|x_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info If *info*=0, the execution is successful. If *info* \neq 0, see [Output Eigensolver info Details](#).

?feast_sbgv/?feast_hbgv

Extended Eigensolver interface for generalized eigenvalue problem with banded matrices.

Syntax

```
void sfeast_sbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
float * a, const MKL_INT * lda, const MKL_INT * klb, const float * b, const MKL_INT *
ldb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float *
emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);

void dfeast_sbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
double * a, const MKL_INT * lda, const MKL_INT * klb, const double * b, const MKL_INT *
ldb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double
* emax, MKL_INT * m0, double * e, double * x, MKL_INT * m, double * res, MKL_INT *
info);

void cfeast_hbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
MKL_Complex8 * a, const MKL_INT * lda, const MKL_INT * klb, const MKL_Complex8 * b,
const MKL_INT * ldb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin,
const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT * m, float * res,
MKL_INT * info);

void zfeast_hbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
MKL_Complex16 * a, const MKL_INT * lda, const MKL_INT * klb, const MKL_Complex16 * b,
const MKL_INT * ldb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double *
emin, const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x, MKL_INT * m,
double * res, MKL_INT * info);
```

Include Files

- mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

NOTE

Both matrices A and B must use the same family of storage format. The bandwidth, however, can be different (klb can be different from kla).

Input Parameters

<i>uplo</i>	Must be 'U' or 'L' or 'F' . If <i>UPLO</i> = 'U', <i>a</i> and <i>b</i> store the upper triangular parts of <i>A</i> and <i>B</i> respectively. If <i>UPLO</i> = 'L', <i>a</i> and <i>b</i> store the lower triangular parts of <i>A</i> and <i>B</i> respectively. If <i>UPLO</i> = 'F', <i>a</i> and <i>b</i> store the full matrices <i>A</i> and <i>B</i> respectively.
<i>n</i>	Sets the size of the problem. $n > 0$.
<i>kla</i>	The number of super- or sub-diagonals within the band in <i>A</i> ($kla \geq 0$).
<i>a</i>	Array of dimension <i>lda</i> by <i>n</i> , contains either full matrix <i>A</i> or upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i>
<i>lda</i>	The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.

<i>klb</i>	The number of super- or sub-diagonals within the band in B ($klb \geq 0$).
<i>b</i>	Array of dimension <i>ldb</i> by <i>n</i> , contains either full matrix B or upper or lower triangular part of the matrix B , as specified by <i>uplo</i>
<i>ldb</i>	The leading dimension of the array B . Must be at least $\max(1, n)$.
<i>fpm</i>	Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin, emax</i>	The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

<i>m0</i>	On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return <i>info</i> =3.
<i>x</i>	On entry, if <i>fpm</i> [4]=1, the array <i>x</i> of size <i>n</i> by <i>m</i> contains a basis of guess subspace where <i>n</i> is the order of the input matrix.

Output Parameters

<i>epsout</i>	On output, contains the relative error on the trace: $ trace_i - trace_{i-1} / \max(emin , emax)$
<i>loop</i>	On output, contains the number of refinement loop executed. Ignored on input.
<i>e</i>	Array of length <i>m0</i> . On output, the first <i>m</i> entries of <i>e</i> are eigenvalues found in the interval.
<i>x</i>	On output, the first <i>m</i> columns of <i>x</i> contain the orthonormal eigenvectors corresponding to the computed eigenvalues <i>e</i> , with the <i>i</i> -th column of <i>x</i> holding the eigenvector associated with $e[i]$.
<i>m</i>	The total number of eigenvalues found in the interval $[emin, emax]$: $0 \leq m \leq m0$.
<i>res</i>	Array of length <i>m0</i> . On exit, the first <i>m</i> components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|Bx_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info

If *info*=0, the execution is successful. If *info* ≠ 0, see [Output Eigensolver info Details](#).

?feast_scsrev/?feast_hcsrev

Extended Eigensolver interface for standard eigenvalue problem with sparse matrices.

Syntax

```
void sfeast_scsrev (const char * uplo, const MKL_INT * n, const float * a, const
MKL_INT * ia, const MKL_INT * ja, MKL_INT * fpm, float * epsout, MKL_INT * loop, const
float * emin, const float * emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float
* res, MKL_INT * info);
```

```
void dfeast_scsrev (const char * uplo, const MKL_INT * n, const double * a, const
MKL_INT * ia, const MKL_INT * ja, MKL_INT * fpm, double * epsout, MKL_INT * loop, const
double * emin, const double * emax, MKL_INT * m0, double * e, double * x, MKL_INT * m,
double * res, MKL_INT * info);
```

```
void cfeast_hcsrev (const char * uplo, const MKL_INT * n, const MKL_Complex8 * a, const
MKL_INT * ia, const MKL_INT * ja, MKL_INT * fpm, float * epsout, MKL_INT * loop, const
float * emin, const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT *
m, float * res, MKL_INT * info);
```

```
void zfeast_hcsrev (const char * uplo, const MKL_INT * n, const MKL_Complex16 * a,
const MKL_INT * ia, const MKL_INT * ja, MKL_INT * fpm, double * epsout, MKL_INT * loop,
const double * emin, const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x,
MKL_INT * m, double * res, MKL_INT * info);
```

Include Files

- mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

uplo

Must be 'U' or 'L' or 'F' .

If *uplo* = 'U', *a* stores the upper triangular parts of *A*.

If *uplo* = 'L', *a* stores the lower triangular parts of *A*.

If *uplo*= 'F' , *a* stores the full matrix *A*.

n

Sets the size of the problem. $n > 0$.

a

Array containing the nonzero elements of either the full matrix *A* or the upper or lower triangular part of the matrix *A*, as specified by *uplo*.

<i>ia</i>	Array of length $n + 1$, containing indices of elements in the array <i>a</i> , such that <i>ia</i> [<i>i</i>] is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> [<i>n</i>] is equal to the number of non-zeros plus one.
<i>ja</i>	Array containing the column indices for each non-zero element of the matrix <i>A</i> being represented in the array <i>a</i> . Its length is equal to the length of the array <i>a</i> .
<i>fpm</i>	Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin, emax</i>	The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

<i>m0</i>	On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where <i>m</i> is the total number of eigenvalues located in the interval [<i>emin</i> , <i>emax</i>]. If the initial guess is wrong, Extended Eigensolver routines return <i>info</i> =3.
<i>x</i>	On entry, if <i>fpm</i> [4]=1, the array <i>x</i> of size <i>n</i> by <i>m</i> contains a basis of guess subspace where <i>n</i> is the order of the input matrix.

Output Parameters

<i>fpm</i>	On output, the last 64 values correspond to Intel® oneAPI Math Kernel Library (oneMKL) PARDISO <i>iparm</i> [0] to <i>iparm</i> [63] (regardless of the value of <i>fpm</i> [63] on input).
<i>epsout</i>	On output, contains the relative error on the trace: $ trace_i - trace_{i-1} / \max(emin , emax)$
<i>loop</i>	On output, contains the number of refinement loop executed. Ignored on input.
<i>e</i>	Array of length <i>m0</i> . On output, the first <i>m</i> entries of <i>e</i> are eigenvalues found in the interval.
<i>x</i>	On output, the first <i>m</i> columns of <i>x</i> contain the orthonormal eigenvectors corresponding to the computed eigenvalues <i>e</i> , with the <i>i</i> -th column of <i>x</i> holding the eigenvector associated with <i>e</i> [<i>i</i>].
<i>m</i>	The total number of eigenvalues found in the interval [<i>emin</i> , <i>emax</i>]: $0 \leq m \leq m0$.
<i>res</i>	Array of length <i>m0</i> . On exit, the first <i>m</i> components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i x_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|x_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info

If *info*=0, the execution is successful. If *info* ≠ 0, see [Output Eigensolver info Details](#).

?feast_scsrgv/?feast_hcsrgv

Extended Eigensolver interface for generalized eigenvalue problem with sparse matrices.

Syntax

```
void sfeast_scsrgv (const char * uplo, const MKL_INT * n, const float * a, const
MKL_INT * ia, const MKL_INT * ja, const float * b, const MKL_INT * ib, const MKL_INT *
jb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float *
emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);

void dfeast_scsrgv (const char * uplo, const MKL_INT * n, const double * a, const
MKL_INT * ia, const MKL_INT * ja, const double * b, const MKL_INT * ib, const MKL_INT *
jb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double *
emax, MKL_INT * m0, double * e, double * x, MKL_INT * m, double * res, MKL_INT * info);

void cfeast_hcsrgv (const char * uplo, const MKL_INT * n, const MKL_Complex8 * a, const
MKL_INT * ia, const MKL_INT * ja, const MKL_Complex8 * b, const MKL_INT * ib, const
MKL_INT * jb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const
float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT * m, float * res,
MKL_INT * info);

void zfeast_hcsrgv (const char * uplo, const MKL_INT * n, const MKL_Complex16 * a,
const MKL_INT * ia, const MKL_INT * ja, const MKL_Complex16 * b, const MKL_INT * ib,
const MKL_INT * jb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double *
emin, const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x, MKL_INT * m,
double * res, MKL_INT * info);
```

Include Files

- mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

NOTE

Both matrices A and B must use the same family of storage format. The position of the non-zero elements can be different (CSR coordinates ib and jb can be different from ia and ja).

Input Parameters

uplo Must be 'U' or 'L' or 'F'.

If $UPLO = 'U'$, a and b store the upper triangular parts of A and B respectively.

If $UPLO = 'L'$, a and b store the lower triangular parts of A and B respectively.

If $UPLO = 'F'$, a and b store the full matrices A and B respectively.

n	Sets the size of the problem. $n > 0$.
a	Array containing the nonzero elements of either the full matrix A or the upper or lower triangular part of the matrix A , as specified by $uplo$.
ia	Array of length $n + 1$, containing indices of elements in the array a , such that $ia[i - 1]$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia[n]$ is equal to the number of non-zeros plus one.
ja	Array containing the column indices for each non-zero element of the matrix A being represented in the array a . Its length is equal to the length of the array a .
b	Array of dimension ldb by $*$, contains the nonzero elements of either the full matrix B or the upper or lower triangular part of the matrix B , as specified by $uplo$.
ib	Array of length $n + 1$, containing indices of elements in the array b , such that $ib[i - 1]$ is the index in the array b of the first non-zero element from the row i . The value of the last element $ib[n]$ is equal to the number of non-zeros plus one.
jb	Array containing the column indices for each non-zero element of the matrix B being represented in the array b . Its length is equal to the length of the array b .
fpm	Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
$emin, emax$	The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.

NOTE Users are advised to avoid situations in which eigenvalues nearly coincide with the interval endpoints. This may lead to unpredictable selection or omission of such eigenvalues. Users should instead specify a slightly larger interval than needed and, if required, pick valid eigenvalues and their corresponding eigenvectors for subsequent use.

$m0$	On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return $info=3$.
x	On entry, if $fpm[4]=1$, the array x of size n by m contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

<i>fpm</i>	On output, the last 64 values correspond to Intel® oneAPI Math Kernel Library (oneMKL) PARDISO <i>iparm</i> [0] to <i>iparm</i> [63] (regardless of the value of <i>fpm</i> [63] on input).
<i>epsout</i>	On output, contains the relative error on the trace: $ trace_i - trace_{i-1} / \max(e_{min} , e_{max})$
<i>loop</i>	On output, contains the number of refinement loop executed. Ignored on input.
<i>e</i>	Array of length <i>m0</i> . On output, the first <i>m</i> entries of <i>e</i> are eigenvalues found in the interval.
<i>x</i>	On output, the first <i>m</i> columns of <i>x</i> contain the orthonormal eigenvectors corresponding to the computed eigenvalues <i>e</i> , with the <i>i</i> -th column of <i>x</i> holding the eigenvector associated with <i>e</i> [<i>i</i>].
<i>m</i>	The total number of eigenvalues found in the interval [<i>e_{min}</i> , <i>e_{max}</i>]: $0 \leq m \leq m0$.
<i>res</i>	<p>Array of length <i>m0</i>. On exit, the first <i>m</i> components contain the relative residual vector:</p> $\frac{\ Ax_i - \lambda_i Bx_i\ _1}{\max(E_{min} , E_{max}) \ Bx_i\ _1}$ <p>for <i>i</i>=1, 2, ..., <i>m</i>, and where <i>m</i> is the total number of eigenvalues found in the search interval.</p>
<i>info</i>	If <i>info</i> =0, the execution is successful. If <i>info</i> ≠ 0, see Output Eigensolver info Details .

Extended Eigensolver Interfaces for Extremal Eigenvalues/Singular Values

The topics in this section discuss Extended Eigensolver interfaces to find extremal eigenvalues as well as singular values.

Extended Eigensolver Interfaces to find largest/smallest eigenvalues

The predefined interfaces include routines for standard and generalized eigenvalue problems and sparse matrices.

Matrix Type	Standard Eigenvalue Problem	Generalized Eigenvalue Problem
Sparse	mkl_sparse?_ev	mkl_sparse?_gv

[mkl_sparse?_ev](#)

Computes the largest/smallest eigenvalues and corresponding eigenvectors of a standard eigenvalue problem

Syntax

```
sparse_status_t mkl_sparse_s_ev (char *which, MKL_INT *pm, sparse_matrix_t A, struct
matrix_descr descrA, MKL_INT k0, MKL_INT *k, float *E, float *X, float *res);
```

```
sparse_status_t mkl_sparse_d_ev (char *which, MKL_INT *pm, sparse_matrix_t A, struct
matrix_descr descrA, MKL_INT k0, MKL_INT *k, double *E, double *X, double *res);
```

Include Files

- mkl_solvers_ee.h

Description

The `mkl_sparse_?_ev` routine computes the largest/smallest eigenvalues and corresponding eigenvectors of a standard eigenvalue problem.

$$Ax = \lambda x$$

where A is the real symmetric matrix.

Input Parameters

which	Indicates eigenvalues for which to search: <ul style="list-style-type: none"> • <code>which = 'L'</code> indicates the largest eigenvalues. • <code>which = 'S'</code> indicates the smallest eigenvalues.
pm	Array of size 128. This array is used to pass various parameters to Extended Eigensolver routines. See • Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem for a complete description of the parameters and their default values.
A	Handle containing sparse matrix in internal data structure.
descrA	Structure specifying sparse matrix properties.
sparse_matrix_type_t type	Specifies the type of a sparse matrix: <ul style="list-style-type: none"> • <code>SPARSE_MATRIX_TYPE_GENERAL</code> <p>The matrix is processed as-is.</p> • <code>SPARSE_MATRIX_TYPE_SYMMETRIC</code> <p>The matrix is symmetric (only the requested triangle is processed).</p>
sparse_fill_mode_t mode	Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices: <ul style="list-style-type: none"> • <code>SPARSE_FILL_MODE_LOWER</code> <p>The lower triangular matrix part is processed.</p> • <code>SPARSE_FILL_MODE_UPPER</code> <p>The upper triangular matrix part is processed.</p>
sparse_diag_type_t diag	Specifies the diagonal type for non-general matrices:

- `SPARSE_DIAG_NON_UNIT`

Diagonal elements might not be equal to one.

- `SPARSE_DIAG_UNIT`

Diagonal elements are equal to one

`k0` The desired number of the largest/smallest eigenvalues to find.

Output Parameters

`k` Number of eigenvalues found.

`E` Array of size `k0`. Contains `k` largest/smallest eigenvalues.

`X` Array of size `k0`*Number of columns of the matrix `A`. Contains `k` eigenvectors.

`Res` Array of size `k0`. Contains `k` residuals.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

`mkl_sparse_?_gv`

Computes the largest/smallest eigenvalues and corresponding eigenvectors of a generalized eigenvalue problem

Syntax

```
sparse_status_t mkl_sparse_s_gv (char *which, MKL_INT *pm, sparse_matrix_t A, struct
matrix_descr descrA, sparse_matrix_t B, struct matrix_descr descrB, MKL_INT k0, MKL_INT
*k, float *E, float *X, float *res);
```

```
sparse_status_t mkl_sparse_d_gv (char *which, MKL_INT *pm, sparse_matrix_t A, struct
matrix_descr descrA, sparse_matrix_t B, struct matrix_descr descrB, MKL_INT k0, MKL_INT
*k, double *E, double *X, double *res);
```

Include Files

- `mkl_solvers_ee.h`

Description

The `mk1_sparse_?_gv` routine computes the largest/smallest eigenvalues and corresponding eigenvectors of a generalized eigenvalue problem.

$$Ax = \lambda Bx$$

where **A** is the real symmetric matrix and **B** is the real symmetric positive definite matrix.

Input Parameters

which	Indicates eigenvalues for which to search: <ul style="list-style-type: none"> • <code>which = 'L'</code> indicates the largest eigenvalues. • <code>which = 'S'</code> indicates the smallest eigenvalues.
pm	Array of size 128. This array is used to pass various parameters to Extended Eigensolver routines. See • Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem for a complete description of the parameters and their default values.
A	Handle containing sparse matrix in internal data structure.
descrA	Structure specifying sparse matrix properties. <div> <div>sparse_matrix_type_t type</div> <div>Specifies the type of a sparse matrix: <ul style="list-style-type: none"> • <code>SPARSE_MATRIX_TYPE_GENERAL</code> The matrix is processed as-is. • <code>SPARSE_MATRIX_TYPE_SYMMETRIC</code> The matrix is symmetric (only the requested triangle is processed). </div> </div> <div> <div>sparse_fill_mode_t mode</div> <div>Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices: <ul style="list-style-type: none"> • <code>SPARSE_FILL_MODE_LOWER</code> The lower triangular matrix part is processed. • <code>SPARSE_FILL_MODE_UPPER</code> The upper triangular matrix part is processed. </div> </div> <div> <div>sparse_diag_type_t diag</div> <div>Specifies the diagonal type for non-general matrices: <ul style="list-style-type: none"> • <code>SPARSE_DIAG_NON_UNIT</code> Diagonal elements might not be equal to one. • <code>SPARSE_DIAG_UNIT</code> Diagonal elements are equal to one </div> </div>
B	Handle containing sparse matrix in internal data structure.
descrB	Structure specifying sparse matrix properties.

<code>sparse_matrix_type_t</code> <i>type</i>	<p>Specifies the type of a sparse matrix:</p> <ul style="list-style-type: none"> ● <code>SPARSE_MATRIX_TYPE_GENERAL</code> The matrix is processed as-is. ● <code>SPARSE_MATRIX_TYPE_SYMMETRIC</code> The matrix is symmetric (only the requested triangle is processed).
<code>sparse_fill_mode_t</code> <i>mode</i>	<p>Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:</p> <ul style="list-style-type: none"> ● <code>SPARSE_FILL_MODE_LOWER</code> The lower triangular matrix part is processed. ● <code>SPARSE_FILL_MODE_UPPER</code> The upper triangular matrix part is processed.
<code>sparse_diag_type_t</code> <i>diag</i>	<p>Specifies the diagonal type for non-general matrices:</p> <ul style="list-style-type: none"> ● <code>SPARSE_DIAG_NON_UNIT</code> Diagonal elements might not be equal to one. ● <code>SPARSE_DIAG_UNIT</code> Diagonal elements are equal to one

`k0` The desired number of the largest/smallest eigenvalues to find.

Output Parameters

<code>k</code>	Number of eigenvalues found.
<code>E</code>	Array of size <code>k0</code> . Contains <code>k</code> largest/smallest eigenvalues.
<code>X</code>	Array of size <code>k0</code> *Number of columns of matrix A. Contains <code>k</code> eigenvectors.
<code>Res</code>	Array of size <code>k0</code> . Contains <code>k</code> residuals.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.
<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

Extended Eigensolver Interfaces to find largest/smallest singular values

The predefined interfaces include routines to find the largest and smallest singular values and the corresponding singular vectors of sparse matrices.

Matrix Type	Standard singular value problem
Sparse	mkl_sparse_?_svd

[mkl_sparse_?_svd](#)

Computes the largest/smallest singular values of a singular-value problem

Syntax

```
sparse_status_t mkl_sparse_s_svd (char *whichS, char *whichV, MKL_INT *pm,
sparse_matrix_t A, struct matrix_descr descrA, MKL_INT k0, MKL_INT *k, float *E, float
*XL, float *XR, float *res);

sparse_status_t mkl_sparse_d_svd (char *whichS, char *whichV, MKL_INT *pm,
sparse_matrix_t A, struct matrix_descr descrA, MKL_INT k0, MKL_INT *k, double *E,
double *XL, double *XR, double *res);
```

Include Files

- `mkl_solvers_ee.h`

Description

The `mkl_sparse_?_svd` routine computes the largest/smallest singular values of a singular-value problem. $AATx = \sigma x$ or $ATAx = \sigma x$, where A is the real rectangular matrix.

Input Parameters

whichS	Indicates eigenvalues for which to search: <ul style="list-style-type: none">• <code>whichS = 'L'</code> indicates the largest eigenvalues.• <code>whichS = 'S'</code> indicates the smallest eigenvalues.
whichV	Indicates singular vectors for which to search: <ul style="list-style-type: none">• <code>whichV = 'R'</code> indicates right singular vectors.• <code>whichV = 'L'</code> indicates left singular vectors.
pm	Array of size 128. This array is used to pass various parameters to Extended Eigensolver routines. See • Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem for a complete description of the parameters and their default values.
A	Handle containing sparse matrix in internal data structure.
descrA	Structure specifying sparse matrix properties. sparse_matrix_type_t Specifies the type of a sparse matrix: type

	<ul style="list-style-type: none"> • <code>SPARSE_MATRIX_TYPE_GENERAL</code> The matrix is processed as-is. • <code>SPARSE_MATRIX_TYPE_SYMMETRIC</code> The matrix is symmetric (only the requested triangle is processed).
<code>sparse_fill_mode_t</code> <i>mode</i>	<p>Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:</p> <ul style="list-style-type: none"> • <code>SPARSE_FILL_MODE_LOWER</code> The lower triangular matrix part is processed. • <code>SPARSE_FILL_MODE_UPPER</code> The upper triangular matrix part is processed.
<code>sparse_diag_type_t</code> <i>diag</i>	<p>Specifies the diagonal type for non-general matrices:</p> <ul style="list-style-type: none"> • <code>SPARSE_DIAG_NON_UNIT</code> Diagonal elements might not be equal to one. • <code>SPARSE_DIAG_UNIT</code> Diagonal elements are equal to one

`k0` The desired number of the largest/smallest eigenvalues to find.

Output Parameters

<code>k</code>	Number of eigenvalues found.
<code>E</code>	Array of size <code>k0</code> . Contains <code>k</code> largest/smallest eigenvalues.
<code>XL</code>	Array of size <code>k0*Number of rows of matrix A</code> . Contains <code>k</code> left singular vectors.
<code>XR</code>	Array of size <code>k0*Number of columns of matrix A</code> . Contains <code>k</code> right singular vectors.
<code>Res</code>	Array that contains <code>k</code> residuals.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<code>SPARSE_STATUS_SUCCESS</code>	The operation was successful.
<code>SPARSE_STATUS_NOT_INITIALIZED</code>	The routine encountered an empty handle or matrix array.
<code>SPARSE_STATUS_ALLOC_FAILED</code>	Internal memory allocation failed.
<code>SPARSE_STATUS_INVALID_VALUE</code>	The input parameters contain an invalid value.
<code>SPARSE_STATUS_EXECUTION_FAILED</code>	Execution failed.

<code>SPARSE_STATUS_INTERNAL_ERROR</code>	An error in algorithm implementation occurred.
<code>SPARSE_STATUS_NOT_SUPPORTED</code>	The requested operation is not supported.

mkl_sparse_ee_init

Initializes Extended Eigensolver input parameters with default values

Syntax

```
sparse_status_t mkl_sparse_ee_init (MKL_INT* pm);
```

Include Files

- `mkl_solvers_ee.h`

Description

This routine sets all Extended Eigensolver parameters to their default values.

Output Parameters

<code>pm</code>	Array of size 128. This array is used to pass various parameters to Extended Eigensolver routines. See • Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem for a complete description of the parameters and their default values.
-----------------	---

Extended Eigensolver Input Parameters for Extremal Eigenvalue Problem

The input parameters for Extended Eigensolver routines are contained in an `MKL_INT` array named `pm`. To call the Extended Eigensolver interfaces, initialize this array using the `mkl_sparse_ee_init` routine.

Parameter	Default	Description
<code>pm[0]</code>	0	Reserved for future use.
<code>pm[1]</code>	6	Defines the tolerance for the stopping criteria: $\epsilon = 10^{-pm[1]} + 1$
<code>pm[2]</code>	0	Specifies the algorithm to use: <ul style="list-style-type: none"> • 0 - Decided at runtime • 1 - The Krylov-Schur method • 2 - Subspace Iteration technique based on FEAST algorithm
<code>pm[3]</code>	*	This parameter is referenced only for Krylov-Schur Method. It indicates the number of Lanczos/Arnoldi vectors (NCV) generated at each iteration. This parameter must be less than or equal to size of matrix and greater than number of eigenvalues (k0) to be computed. If unspecified, NCV is set to be at least 1.5 times larger than k0.
<code>pm[4]</code>	*	Maximum number of iterations. If unspecified, this parameter is set to 10000 for the Krylov-Schur method and 60 for the subspace iteration method.

Parameter	Default	Description
<code>pm[5]</code>	0	Power of Chebychev expansion for approximate spectral projector. Only referenced when <code>pm[2]=1</code>
<code>pm[6]</code>	1	Used only for Krylov-Schur Method. If 0, then the method only computes eigenvalues. If 1, then the method computes eigenvalues and eigenvectors. The subspace iteration method always computes eigenvectors/singular vectors. You must allocate the required memory space.
<code>pm[7]</code>	0	Convergence stopping criteria. Defines whether the stopping criteria for the iterations with respect to the true residuals (used if <code>pm[8]</code> is not zero) and residual norm estimates are relative to the eigenvalues/singular values or not. If 0, the stopping criteria with respect to the true residuals is: $\frac{\ Ax - \lambda x\ }{ \lambda } < 10^{-pm[1]} + 1$ or $\frac{\ Ax - \lambda Bx\ }{ \lambda } < 10^{-pm[1]} + 1$ If 1, the stopping criteria with respect to the true residuals is: $\ Ax - \lambda x\ < 10^{-pm[1]} + 1$ or $\ Ax - \lambda Bx\ < 10^{-pm[1]} + 1$ for a generalized eigenproblem. The residual norm estimates are based on the magnitude of the last eigenvector of the Schur decomposition matrix and the exact formula can be found in the literature. When <code>pm[7]=0</code> , the residual norm estimate is additionally divided by the magnitude of the computed eigenvalue and compared to $10^{(-pm[1]+1)}$.
<code>pm[8]</code>	0	Specifies if for detecting convergence the solver must compute the true residuals for eigenpairs for the Krylov-Schur method or it can only use the residual norm estimates. If 0, only residual norm estimates are used. If 1, the solver computes not just residual norm estimates but also the true residuals as defined in the description of <code>pm[7]</code> .
<code>pm[9]</code>	0	Used only for the Krylov-Schur method and only as an output parameter. Reports the reason for exiting the iteration loop of the method: <ul style="list-style-type: none"> • If 0, the iterations stopped since convergence has been detected. • If -1, maximum number of iterations has been reached and even the residual norm estimates have not converged.

Parameter	Default	Description
		<ul style="list-style-type: none"> • If -2, maximum number of iterations has been reached despite the residual norm estimates have converged (but the true residuals for eigenpairs have not). • If -3, the iterations stagnated and even the residual norm estimates have not converged. • If -4, the iterations stagnated while the eigenvalues have converged (but the true residuals for eigenpairs do not).
<code>pm[10]</code> to <code>pm[128]</code>	-	Reserved for future use.

Vector Mathematical Functions

Intel® oneAPI Math Kernel Library (oneMKL) Vector Mathematics functions (VM) compute a mathematical function of each of the vector elements. VM includes a set of highly optimized functions (arithmetic, power, trigonometric, exponential, hyperbolic, special, and rounding) that operate on vectors of real and complex numbers.

Application programs that improve performance with VM include nonlinear programming software, computation of integrals, financial calculations, computer graphics, and many others.

VM functions fall into the following groups according to the operations they perform:

- [VM Mathematical Functions](#) compute values of mathematical functions, such as sine, cosine, exponential, or logarithm, on vectors stored contiguously in memory.
- [VM Pack/Unpack Functions](#) convert to and from vectors with positive increment indexing, vector indexing, and mask indexing (see [Appendix "Vector Arguments in VM"](#) for details on vector indexing methods).
- [VM Service Functions](#) set/get the accuracy modes and the error codes, and free memory.

The VM mathematical functions take an input vector as an argument, compute values of the respective function element-wise, and return the results in an output vector. All the VM mathematical functions can perform in-place operations, where the input and output arrays are at the same memory locations. For VM mathematical functions with positive increment indexing, in-place operations are supported only when the input and output increments have the same value.

The Intel® oneAPI Math Kernel Library (oneMKL) interfaces are given in `mkl_vml_functions.h`.

NOTE On 64-bit platforms, oneMKL provides VML C interfaces with the `_64` suffix to support large data arrays in the LP64 interface library. For more interface library details, see "Using the ILP64 Interface vs. LP64 Interface" in the developer guide.

Examples that demonstrate how to use the VM functions are located in:

`${MKL}/examples/vmlc/source`

See VM performance and accuracy data in the online VM Performance and Accuracy Data document available at <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html>.

Product and Performance Information

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

Notice revision #20201201

VM Data Types, Accuracy Modes, and Performance Tips

VM includes mathematical and pack/unpack vector functions for single and double precision vector arguments of real and complex types. Intel® oneAPI Math Kernel Library (oneMKL) provides Fortran and C interfaces for all VM functions, including the associated service functions. The Function Naming Conventions topic shows how to call these functions.

Performance depends on a number of factors, including vectorization and threading overhead. The recommended usage is as follows:

- Use VM for vector lengths larger than 40 elements.
- Use the Intel® Compiler for vector lengths less than 40 elements.

All VM vector functions support the following accuracy modes:

- High Accuracy (HA), the default mode
- Low Accuracy (LA), which improves performance by reducing accuracy of the two least significant bits
- Enhanced Performance (EP), which provides better performance at the cost of significantly reduced accuracy. Approximately half of the bits in the mantissa are correct.

Note that using the EP mode does not guarantee accurate processing of corner cases and special values. Although the default accuracy is HA, LA is sufficient in most cases. For applications that require less accuracy (for example, media applications, some Monte Carlo simulations, etc.), the EP mode may be sufficient.

VM handles special values in accordance with the C99 standard [C99].

Intel® oneAPI Math Kernel Library (oneMKL) offers both functions and environment variables to switch between modes for VM. See the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details about the environment variables. Use the `vm1SetMode(mode)` function (see [Table "Values of the mode Parameter"](#)) to switch between the HA, LA, and EP modes. The `vm1GetMode()` function returns the current mode.

Product and Performance Information

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

Notice revision #20201201

See Also

[Function Naming Conventions](#)

VM Naming Conventions

The VM function names are of mixed (lower and upper) case.

The VM mathematical and pack/unpack function names have the following structure:

`v[m]<?><name><mod>`

where

- `v` is a prefix indicating vector operations.
- `[m]` is an optional prefix for mathematical functions that indicates additional argument to specify a VM mode for a given function call (see [vm1SetMode](#) for possible values and their description).
- `<?>` is a precision prefix that indicates one of the following data types:

<code>s</code>	<code>float.</code>
<code>d</code>	<code>double.</code>
<code>c</code>	<code>MKL_Complex8.</code>
<code>z</code>	<code>MKL_Complex16.</code>

- `<name>` indicates the function short name, with some of its letters in uppercase. See examples in [Table "VM Mathematical Functions"](#).
- `<mod>` field (written in uppercase) is present only in the pack/unpack functions and indicates the indexing method used:

<i>i</i>	indexing with a positive increment
<i>v</i>	indexing with an index vector
<i>m</i>	indexing with a mask vector.

The VM service function names have the following structure:

`vm<name>`

where

`<name>` indicates the function short name, with some of its letters in uppercase. See examples in [Table "VM Service Functions"](#).

To call VM functions from an application program, use conventional function calls. For example, call the vector single precision real exponential function as

```
vsExp ( n, a, y );
```

VM Function Interfaces

VM interfaces include the function names and argument lists. The following sections describe the interfaces for the VM functions. Note that some of the functions have multiple input and output arguments

Some VM functions may also take scalar arguments as input. See the function description for the naming conventions of such arguments.

VM Mathematical Function Interfaces

```
v<?><name>( n, a, [scalar input arguments], y );
v<?><name>I( n, a, inca, [scalar input arguments], y, incy );
v<?><name>( n, a, b, [scalar input arguments], y );
v<?><name>I( n, a, inca, b, incb, [scalar input arguments], y, incy );
v<?><name>( n, a, y, z );
v<?><name>I( n, a, inca, y, incy, z, incz );
vm<?><name>( n, a, [scalar input arguments], y, mode );
vm<?><name>I( n, a, inca, [scalar input arguments], y, incy, mode );
vm<?><name>( n, a, b, [scalar input arguments], y, mode );
vm<?><name>I( n, a, inca, b, incb, [scalar input arguments], y, incy, mode );
vm<?><name>( n, a, y, z, mode );
vm<?><name>I( n, a, inca, y, incy, z, incz, mode );
```

VM Mathematical Functions

VM Pack Function Interfaces

```
v<?>PackI( n, a, inca, y );
v<?>PackV( n, a, ia, y );
v<?>PackM( n, a, ma, y );
```

VM Unpack Function Interfaces

```
v<?>UnpackI( n, a, y, incy );
v<?>UnpackV( n, a, y, iy );
```

```
v<?>UnpackM( n, a, y, my );
```

VM Service Function Interfaces

```
oldmode = vmlSetMode( mode );
mode = vmlGetMode( void );
olderr = vmlSetErrStatus ( err );
err = vmlGetErrStatus( void );
olderr = vmlClearErrStatus( void );
oldcallback = vmlSetErrorCallBack( callback );
callback = vmlGetErrorCallBack( void );
oldcallback = vmlClearErrorCallBack( void );
```

Note that *oldmode*, *olderr*, and *oldcallback* refer to settings prior to the call.

VM Input Parameters

<i>n</i>	number of elements to be calculated
<i>a</i>	first input vector
<i>b</i>	second input vector
<i>inca</i>	vector increment for the input vector <i>a</i>
<i>incb</i>	vector increment for the input vector <i>b</i>
<i>ia</i>	index vector for the input vector <i>a</i>
<i>ma</i>	mask vector for the input vector <i>a</i>
<i>incy</i>	vector increment for the output vector <i>y</i>
<i>incz</i>	vector increment for the output vector <i>z</i>
<i>iy</i>	index vector for the output vector <i>y</i>
<i>my</i>	mask vector for the output vector <i>y</i>
<i>err</i>	error code
<i>mode</i>	VM mode
<i>callback</i>	address of the callback function

VM Output Parameters

<i>y</i>	first output vector
<i>z</i>	second output vector
<i>err</i>	error code
<i>mode</i>	VM mode
<i>olderr</i>	former error code
<i>oldmode</i>	former VM mode
<i>callback</i>	address of the callback function
<i>oldcallback</i>	address of the former callback function

See the data types of the parameters used in each function in the respective function description section. All Intel® oneAPI Math Kernel Library (oneMKL) VM mathematical functions can perform in-place operations. For VM mathematical functions with positive increment indexing, (for example, `v?PowI`), in-place operations are supported only when the input and output increments have the same value.

Vector Indexing Methods

Classic VM mathematical functions work with unit stride. Strided VM mathematical functions (names with "I" suffix) work with arbitrary integer increments. Increments may be positive, negative or equal to zero. For example:

```
vsExpI (n, a, inca, r, incr)
```

is equivalent to:

```
for (i=0; i<n; i++)
{
    r[i * incr] = exp (a[i * inca]);
}
```

where

i – current index,

inca – input index increment,

incr – output index increment.

n – the number of elements to be computed (important: *n* is not the maximum array size).

So, when calling `vsExpI(n, a, inca, r, incr)` be sure that the input vector *a* is allocated at least for $1 + (n-1)*inca$ elements and the result vector *r* has a space for $1 + (n-1)*incr$ elements.

NOTE The order of computations is not guaranteed and no array bounds-checking is performed; therefore, the results for overlapped and in-place arrays are not generally deterministic for increments other than 1.

For output index increment, equal to 0, the result is not deterministic and generally nonsensical.

Use negative increments to step from base pointers in reverse order.

For example:

```
vsExpI (n, a, -2, r, -3)
```

is equivalent to:

```
for (i=0; i<n; i++)
{
    r[- i*3] = exp (a[-i*2]);
}
```

NOTE Pass pointers to the desired ending array element in memory as an argument for negative strides.

For example:

```
vsExpI (n, a, 2, r + 1000, -3).
```

Use a zero increment for one fixed argument rather than an array.

For example:

```
vsMulI (n, a, 1, b, 0, r, 1)
```

is equivalent to:

```
for (i=0; i<n; i++)
{
    r[i] = a[i] * b[0];
}
```

VM Pack/Unpack functions use the following indexing methods to do this task:

- positive increment
- index vector
- mask vector

The indexing method used in a particular function is indicated by the indexing modifier (see the description of the `<mod>` field in [Function Naming Conventions](#)). For more information on the indexing methods, see [Vector Arguments in VM](#).

VM Pack/Unpack Functions

VM Error Diagnostics

The VM mathematical functions incorporate the error handling mechanism, which is controlled by the following service functions:

`vmlGetErrStatus`,
`vmlSetErrStatus`,
`vmlClearErrStatus`

These functions operate with a global variable called VM Error Status. The VM Error Status flags an error, a warning, or a successful execution of a VM function.

`vmlGetErrCallBack`,
`vmlSetErrCallBack`,
`vmlClearErrCallBack`

These functions enable you to customize the error handling. For example, you can identify a particular argument in a vector where an error occurred or that caused a warning.

`vmlSetMode`, `vmlGetMode`

These functions get and set a VM mode. If you set a new VM mode using the `vmlSetMode` function, you can store the previous VM mode returned by the routine and restore it at any point of your application.

If both an error and a warning situation occur during the function call, the VM Error Status variable keeps only the value of the error code. See [Table "Values of the VM Error Status"](#) for possible values. If a VM function does not encounter errors or warnings, it sets the VM Error Status to `VML_STATUS_OK`.

If you use incorrect input arguments to a VM function (`VML_STATUS_BADSIZE` and `VML_STATUS_BADMEM`), the function calls `xerbla` to report errors. See [Table "Values of the VM Error Status"](#) for details.

You can use the `vmlSetMode` and `vmlGetMode` functions to modify error handling behavior. Depending on the VM mode, the error handling behavior includes the following operations:

- setting the VM Error Status to a value corresponding to the observed error or warning
- setting the `errno` variable to one of the values described in [Table "Set Values of the `errno` Variable"](#)
- writing error text information to the `stderr` stream
- raising the appropriate exception on an error, if necessary
- calling the additional error handler callback function that is set by `vmlSetErrorCallBack`.

Set Values of the `errno` Variable

Value of <code>errno</code>	Description
0	No errors are detected.
<code>EINVAL</code>	The array dimension is not positive.
<code>EACCES</code>	NULL pointer is passed.
<code>EDOM</code>	At least one of array values is out of a range of definition.
<code>ERANGE</code>	At least one of array values caused a singularity, overflow or underflow.

See Also

`vmlGetErrStatus` Gets the VM Error Status.

`vmlSetErrStatus` Sets the new VM Error Status according to *err* and stores the previous VM Error Status to *olderr*. Sets the global VM Status according to new values and returns the previous VM Status.

`vmlClearErrStatus` Sets the VM Error Status to `VML_STATUS_OK` and stores the previous VM Error Status to *olderr*.

`vmlSetErrorCallBack` Sets the additional error handler callback function and gets the old callback function.

`vmlGetErrorCallBack` Gets the additional error handler callback function.

`vmlClearErrorCallBack` Deletes the additional error handler callback function and retrieves the former callback function.

`vmlGetMode` Gets the VM mode.

`vmlSetMode` Sets a new mode for VM functions according to the *mode* parameter and stores the previous VM mode to *oldmode*.

VM Mathematical Functions

This section describes VM functions that compute values of mathematical functions on real and complex vector arguments.

Each function is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data, as well as a description of the input/output arguments.

The input range of parameters is equal to the mathematical range of the input data type, unless the function description specifies input threshold values, which mark off the precision overflow, as follows:

- `FLT_MAX` denotes the maximum number representable in single precision real data type
- `DBL_MAX` denotes the maximum number representable in double precision real data type

Table "VM Mathematical Functions" lists available mathematical functions and associated data types.

VM Mathematical Functions

Function	Data Types	Description
Arithmetic Functions		
<code>v?Add</code>	<i>s, d, c, z</i>	Adds vector elements
<code>v?Sub</code>	<i>s, d, c, z</i>	Subtracts vector elements
<code>v?Sqr</code>	<i>s, d</i>	Squares vector elements
<code>v?Mul</code>	<i>s, d, c, z</i>	Multiplies vector elements
<code>v?MulByConj</code>	<i>c, z</i>	Multiplies elements of one vector by conjugated elements of the second vector
<code>v?Conj</code>	<i>c, z</i>	Conjugates vector elements
<code>v?Abs</code>	<i>s, d, c, z</i>	Computes the absolute value of vector elements
<code>v?Arg</code>	<i>c, z</i>	Computes the argument of vector elements
<code>v?LinearFrac</code>	<i>s, d</i>	Performs linear fraction transformation of vectors
<code>v?Fmod</code>	<i>s, d</i>	Performs element by element computation of the modulus function of vector <i>a</i> with respect to vector <i>b</i>
<code>v?Remainder</code>	<i>s, d</i>	Performs element by element computation of the remainder function on the elements of vector <i>a</i> and the corresponding elements of vector <i>b</i>
Power and Root Functions		
<code>v?Inv</code>	<i>s, d</i>	Inverts vector elements
<code>v?Div</code>	<i>s, d, c, z</i>	Divides elements of one vector by elements of the second vector
<code>v?Sqrt</code>	<i>s, d, c, z</i>	Computes the square root of vector elements
<code>v?InvSqrt</code>	<i>s, d</i>	Computes the inverse square root of vector elements
<code>v?Cbrrt</code>	<i>s, d</i>	Computes the cube root of vector elements

Function	Data Types	Description
v?InvCbrt	<i>s, d</i>	Computes the inverse cube root of vector elements
v?Pow2o3	<i>s, d</i>	Computes the cube root of the square of each vector element
v?Pow3o2	<i>s, d</i>	Computes the square root of the cube of each vector element
v?Pow	<i>s, d, c, z</i>	Raises each vector element to the specified power
v?Powx	<i>s, d, c, z</i>	Raises each vector element to the constant power
v?Powr	<i>s, d</i>	Computes <i>a</i> to the power <i>b</i> for elements of two vectors, where the elements of vector argument <i>a</i> are all non-negative
v?Hypot	<i>s, d</i>	Computes the square root of sum of squares
Exponential and Logarithmic Functions		
v?Exp	<i>s, d, c, z</i>	Computes the base <i>e</i> exponential of vector elements
v?Exp2	<i>s, d</i>	Computes the base 2 exponential of vector elements
v?Exp10	<i>s, d</i>	Computes the base 10 exponential of vector elements
v?Expm1	<i>s, d</i>	Computes the base <i>e</i> exponential of vector elements decreased by 1
v?Ln	<i>s, d, c, z</i>	Computes the natural logarithm of vector elements
v?Log2	<i>s, d</i>	Computes the base 2 logarithm of vector elements
v?Log10	<i>s, d, c, z</i>	Computes the base 10 logarithm of vector elements
v?Log1p	<i>s, d</i>	Computes the natural logarithm of vector elements that are increased by 1
v?Logb	<i>s, d</i>	Computes the exponents of the elements of input vector <i>a</i>
Trigonometric Functions		
v?Cos	<i>s, d, c, z</i>	Computes the cosine of vector elements
v?Sin	<i>s, d, c, z</i>	Computes the sine of vector elements
v?SinCos	<i>s, d</i>	Computes the sine and cosine of vector elements
v?CIS	<i>c, z</i>	Computes the complex exponent of vector elements (cosine and sine combined to complex value)
v?Tan	<i>s, d, c, z</i>	Computes the tangent of vector elements
v?Acos	<i>s, d, c, z</i>	Computes the inverse cosine of vector elements
v?Asin	<i>s, d, c, z</i>	Computes the inverse sine of vector elements
v?Atan	<i>s, d, c, z</i>	Computes the inverse tangent of vector elements
v?Atan2	<i>s, d</i>	Computes the four-quadrant inverse tangent of ratios of the elements of two vectors
v?Cospi	<i>s, d</i>	Computes the cosine of vector elements multiplied by <i>n</i>
v?Sinpi	<i>s, d</i>	Computes the sine of vector elements multiplied by <i>n</i>
v?Tanpi	<i>s, d</i>	Computes the tangent of vector elements multiplied by <i>n</i>
v?Acospi	<i>s, d</i>	Computes the inverse cosine of vector elements divided by <i>n</i>
v?Asinpi	<i>s, d</i>	Computes the inverse sine of vector elements divided by <i>n</i>
v?Atanpi	<i>s, d</i>	Computes the inverse tangent of vector elements divided by <i>n</i>
v?Atan2pi	<i>s, d</i>	Computes the four-quadrant inverse tangent of the ratios of the corresponding elementss of two vectors divided by <i>n</i>
v?Cosd	<i>s, d</i>	Computes the cosine of vector elements multiplied by <i>n</i> /180
v?Sind	<i>s, d</i>	Computes the sine of vector elements multiplied by <i>n</i> /180
v?Tand	<i>s, d</i>	Computes the tangent of vector elements multiplied by <i>n</i> /180
Hyperbolic Functions		
v?Cosh	<i>s, d, c, z</i>	Computes the hyperbolic cosine of vector elements
v?Sinh	<i>s, d, c, z</i>	Computes the hyperbolic sine of vector elements
v?Tanh	<i>s, d, c, z</i>	Computes the hyperbolic tangent of vector elements
v?Acosh	<i>s, d, c, z</i>	Computes the inverse hyperbolic cosine of vector elements
v?Asinh	<i>s, d, c, z</i>	Computes the inverse hyperbolic sine of vector elements
v?Atanh	<i>s, d, c, z</i>	Computes the inverse hyperbolic tangent of vector elements.
Special Functions		
v?Erf	<i>s, d</i>	Computes the error function value of vector elements

Function	Data Types	Description
<code>v?Erfc</code>	<i>s, d</i>	Computes the complementary error function value of vector elements
<code>v?CdfNorm</code>	<i>s, d</i>	Computes the cumulative normal distribution function value of vector elements
<code>v?ErfInv</code>	<i>s, d</i>	Computes the inverse error function value of vector elements
<code>v?ErfcInv</code>	<i>s, d</i>	Computes the inverse complementary error function value of vector elements
<code>v?CdfNormInv</code>	<i>s, d</i>	Computes the inverse cumulative normal distribution function value of vector elements
<code>v?LGamma</code>	<i>s, d</i>	Computes the natural logarithm for the absolute value of the gamma function of vector elements
<code>v?TGamma</code>	<i>s, d</i>	Computes the gamma function of vector elements
<code>v?ExpInt1</code>	<i>s, d</i>	Computes the exponential integral of vector elements
Rounding Functions		
<code>v?Floor</code>	<i>s, d</i>	Rounds towards minus infinity
<code>v?Ceil</code>	<i>s, d</i>	Rounds towards plus infinity
<code>v?Trunc</code>	<i>s, d</i>	Rounds towards zero infinity
<code>v?Round</code>	<i>s, d</i>	Rounds to nearest integer
<code>v?NearbyInt</code>	<i>s, d</i>	Rounds according to current mode
<code>v?Rint</code>	<i>s, d</i>	Rounds according to current mode and raising inexact result exception
<code>v?Modf</code>	<i>s, d</i>	Computes the integer and fractional parts
<code>v?Frac</code>	<i>s, d</i>	Computes the fractional part
Miscellaneous Functions		
<code>v?CopySign</code>	<i>s, d</i>	Returns vector of elements of one argument with signs changed to match other argument elements
<code>v?NextAfter</code>	<i>s, d</i>	Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector
<code>v?Fdim</code>	<i>s, d</i>	Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise
<code>v?Fmax</code>	<i>s, d</i>	Returns the larger of each pair of elements of the two vector arguments
<code>v?Fmin</code>	<i>s, d</i>	Returns the smaller of each pair of elements of the two vector arguments
<code>v?MaxMag</code>	<i>s, d</i>	Returns the element with the larger magnitude between each pair of elements of the two vector arguments
<code>v?MinMag</code>	<i>s, d</i>	Returns the element with the smaller magnitude between each pair of elements of the two vector arguments

Special Value Notations

This topic defines notations of special values for complex functions. The definitions are provided in text, tables, or formulas.

- *z*, *z1*, *z2*, etc. denote complex numbers.
- *i*, $i^2=-1$ is the imaginary unit.
- *x*, *X*, *x1*, *x2*, etc. denote real imaginary parts.
- *y*, *Y*, *y1*, *y2*, etc. denote imaginary parts.
- *X* and *Y* represent any finite positive IEEE-754 floating point values, if not stated otherwise.
- Quiet NaN and signaling NaN are denoted with QNAN and SNAN, respectively.
- The IEEE-754 positive infinities or floating-point numbers are denoted with a + sign before *x*, *y*, etc.
- The IEEE-754 negative infinities or floating-point numbers are denoted with a – sign before *x*, *y*, etc.

CONJ(*z*) and CIS(*z*) are defined as follows:

$$\text{CONJ}(x+i\cdot y)=x-i\cdot y$$

$\text{CIS}(y) = \cos(y) + i \cdot \sin(y)$.

The special value tables show the result of the function for the z argument at the intersection of the $\text{RE}(z)$ column and the $i \cdot \text{IM}(z)$ row. If the function raises an exception on the argument z , the lower part of this cell shows the raised exception and the VM Error Status. An empty cell indicates that this argument is normal and the result is defined mathematically.

Arithmetic Functions

Arithmetic functions perform the basic mathematical operations like addition, subtraction, multiplication or computation of the absolute value of the vector elements.

v?Add

Performs element by element addition of vector a and vector b .

Syntax

```
vsAdd( n, a, b, y );
vsAddI( n, a, inca, b, incb, y, incy );
vmsAdd( n, a, b, y, mode );
vmsAddI( n, a, inca, b, incb, y, incy, mode );
vdAdd( n, a, b, y );
vdAddI( n, a, inca, b, incb, y, incy );
vmdAdd( n, a, b, y, mode );
vmdAddI( n, a, inca, b, incb, y, incy, mode );
vcAdd( n, a, b, y );
vcAddI( n, a, inca, b, incb, y, incy );
vmcAdd( n, a, b, y, mode );
vmcAddI( n, a, inca, b, incb, y, incy, mode );
vzAdd( n, a, b, y );
vzAddI( n, a, inca, b, incb, y, incy );
vmzAdd( n, a, b, y, mode );
vmzAddI( n, a, inca, b, incb, y, incy, mode );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
n	const MKL_INT	Specifies the number of elements to be calculated.
a, b	const float* for vsAdd, vmsAdd const double* for vdAdd, vmdAdd	Pointers to arrays that contain the input vectors a and b .

Name	Type	Description
	const MKL_Complex8* for vcAdd, vmcAdd	
	const MKL_Complex16* for vzAdd, vmzAdd	
<i>inca</i> , <i>incb</i> , <i>incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsAdd, vmsAdd double* for vdAdd, vmdAdd MKL_Complex8* for vcAdd, vmcAdd MKL_Complex16* for vzAdd, vmzAdd	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Add` function performs element by element addition of vector *a* and vector *b*.

Special values for Real Function v?Add

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	+0	
-0	+0	+0	
-0	-0	-0	
$+\infty$	$+\infty$	$+\infty$	
$+\infty$	$-\infty$	QNAN	INVALID
$-\infty$	$+\infty$	QNAN	INVALID
$-\infty$	$-\infty$	$-\infty$	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Add}(x1+i*y1, x2+i*y2) = (x1+x2) + i*(y1+y2)$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all $\text{RE}(x)$, $\text{RE}(y)$, $\text{IM}(x)$, $\text{IM}(y)$ arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

v?Sub

Performs element by element subtraction of vector b from vector a .

Syntax

```
vsSub( n, a, b, y );
vsSubI(n, a, inca, b, incb, y, incy);
vmsSub( n, a, b, y, mode );
vmsSubI(n, a, inca, b, incb, y, incy, mode);
vdSub( n, a, b, y );
vdSubI(n, a, inca, b, incb, y, incy);
vmdSub( n, a, b, y, mode );
vmdSubI(n, a, inca, b, incb, y, incy, mode);
vcSub( n, a, b, y );
vcSubI(n, a, inca, b, incb, y, incy);
vmcSub( n, a, b, y, mode );
vmcSubI(n, a, inca, b, incb, y, incy, mode);
vzSub( n, a, b, y );
vzSubI(n, a, inca, b, incb, y, incy);
vmzSub( n, a, b, y, mode );
vmzSubI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
n	const MKL_INT	Specifies the number of elements to be calculated.
a, b	const float* for vsSub, vmsSub const double* for vdSub, vmdSub const MKL_Complex8* for vcSub, vmcSub const MKL_Complex16* for vzSub, vmzSub	Pointers to arrays that contain the input vectors a and b .
$inca, incb, incy$	const MKL_INT	Specifies increments for the elements of a , b , and y .
$mode$	const MKL_INT64	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	float* for vsSub, vmsSub double* for vdSub, vmdSub MKL_Complex8* for vcSub, vmcSub MKL_Complex16* for vzSub, vmzSub	Pointer to an array that contains the output vector y .

Description

The $v?Sub$ function performs element by element subtraction of vector b from vector a .

Special values for Real Function $v?Sub(x)$

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	+0	
-0	+0	-0	
-0	-0	+0	
$+\infty$	$+\infty$	QNAN	INVALID
$+\infty$	$-\infty$	$+\infty$	
$-\infty$	$+\infty$	$-\infty$	
$-\infty$	$-\infty$	QNAN	INVALID
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Sub}(x1+i*y1, x2+i*y2) = (x1-x2) + i*(y1-y2).$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all $\text{RE}(x)$, $\text{RE}(y)$, $\text{IM}(x)$, $\text{IM}(y)$ arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

$v?Sqr$

Performs element by element squaring of the vector.

Syntax

```
vsSqr( n, a, y );
vsSqrI(n, a, inca, y, incy);
vmsSqr( n, a, y, mode );
vmsSqrI(n, a, inca, y, incy, mode);
vdSqr( n, a, y );
vdSqrI(n, a, inca, y, incy);
vmdSqr( n, a, y, mode );
```

```
vmdSqrI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsSqr</code> , <code>vmsSqr</code> <code>const double*</code> for <code>vdSqr</code> , <code>vmdSqr</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsSqr</code> , <code>vmsSqr</code> <code>double*</code> for <code>vdSqr</code> , <code>vmdSqr</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Sqr` function performs element by element squaring of the vector.

Special Values for Real Function `v?Sqr(x)`

Argument	Result	Exception
+0	+0	
-0	+0	
$+\infty$	$+\infty$	
$-\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	INVALID

`v?Mul`

*Performs element by element multiplication of vector *a* and vector *b*.*

Syntax

```
vsMul( n, a, b, y );
vsMulI(n, a, inca, b, incb, y, incy);
vmsMul( n, a, b, y, mode );
vmsMulI(n, a, inca, b, incb, y, incy, mode);
vdMul( n, a, b, y );
```

```

vdMulI(n, a, inca, b, incb, y, incy);vdMulI(n, a, inca, b, incb, y, incy);
vmdMul( n, a, b, y, mode );
vmdMulI(n, a, inca, b, incb, y, incy, mode);
vcMul( n, a, b, y );
vcMulI(n, a, inca, b, incb, y, incy);
vmcMul( n, a, b, y, mode );
vmcMulI(n, a, inca, b, incb, y, incy, mode);
vzMul( n, a, b, y );
vzMulI(n, a, inca, b, incb, y, incy);
vmzMul( n, a, b, y, mode );
vmzMulI(n, a, inca, b, incb, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	const float* for vsMul, vmsMul const double* for vdMul, vmdMul const MKL_Complex8* for vcMul, vmcMul const MKL_Complex16* for vzMul, vmzMul	Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsMul, vmsMul double* for vdMul, vmdMul MKL_Complex8* for vcMul, vmcMul MKL_Complex16* for vzMul, vmzMul	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Mul` function performs element by element multiplication of vector *a* and vector *b*.

Special values for Real Function v?Mul(x)

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	-0	
-0	+0	-0	
-0	-0	+0	
+0	$+\infty$	QNAN	INVALID
+0	$-\infty$	QNAN	INVALID
-0	$+\infty$	QNAN	INVALID
-0	$-\infty$	QNAN	INVALID
$+\infty$	+0	QNAN	INVALID
$+\infty$	-0	QNAN	INVALID
$-\infty$	+0	QNAN	INVALID
$-\infty$	-0	QNAN	INVALID
$+\infty$	$+\infty$	$+\infty$	
$+\infty$	$-\infty$	$-\infty$	
$-\infty$	$+\infty$	$-\infty$	
$-\infty$	$-\infty$	$+\infty$	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Mul}(x1+i*y1, x2+i*y2) = (x1*x2-y1*y2) + i*(x1*y2+y1*x2).$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all $\text{RE}(x)$, $\text{RE}(y)$, $\text{IM}(x)$, $\text{IM}(y)$ arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the OVERFLOW exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

v?MulByConj

Performs element by element multiplication of vector a element and conjugated vector b element.

Syntax

```
vcMulByConj( n, a, b, y );
vsMulByConjI(n, a, inca, b, incb, y, incy);
vmcMulByConj( n, a, b, y, mode );
vmsMulByConjI(n, a, inca, b, incb, y, incy, mode);
vzMulByConj( n, a, b, y );
vdMulByConjI(n, a, inca, b, incb, y, incy);
vmzMulByConj( n, a, b, y, mode );
vmdMulByConjI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a, b</i>	<code>const MKL_Complex8*</code> for <code>vcMulByConj</code> , <code>vmcMulByConj</code> <code>const MKL_Complex16*</code> for <code>vzMulByConj</code> , <code>vmzMulByConj</code>	Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>MKL_Complex8*</code> for <code>vcMulByConj</code> , <code>vmcMulByConj</code> <code>MKL_Complex16*</code> for <code>vzMulByConj</code> , <code>vmzMulByConj</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?MulByConj` function performs element by element multiplication of vector *a* element and conjugated vector *b* element.

Specifications for special values of the functions are found according to the formula

$$\text{MulByConj}(x1+i*y1, x2+i*y2) = \text{Mul}(x1+i*y1, x2-i*y2).$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all $\text{RE}(x)$, $\text{RE}(y)$, $\text{IM}(x)$, $\text{IM}(y)$ arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

v?Conj

Performs element by element conjugation of the vector.

Syntax

```
vcConj( n, a, y );
vcConjI(n, a, inca, y, incy);
vmcConj( n, a, y, mode );
```

```

vmcConjI(n, a, inca, y, incy, mode);
vzConj( n, a, y );
vzConjI(n, a, inca, y, incy);
vmzConj( n, a, y, mode );
vmzConjI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const MKL_Complex8* for vcConj, vmcConj const MKL_Complex16* for vzConj, vmzConj	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	MKL_Complex8* for vcConj, vmcConj MKL_Complex16* for vzConj, vmzConj	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Conj` function performs element by element conjugation of the vector.

No special values are specified. The function does not raise floating-point exceptions.

v?Abs

Computes absolute value of vector elements.

Syntax

```

vsAbs( n, a, y );
vsAbsI(n, a, inca, y, incy);
vmsAbs( n, a, y, mode );
vmsAbsI(n, a, inca, y, incy, mode);
vdAbs( n, a, y );
vdAbsI(n, a, inca, y, incy);

```



```

vmdAbs( n, a, y, mode );
vmdAbsI(n, a, inca, y, incy, mode);
vcAbs( n, a, y );
vcAbsI(n, a, inca, y, incy);
vmcAbs( n, a, y, mode );
vmcAbsI(n, a, inca, y, incy, mode);
vzAbs( n, a, y );
vzAbsI(n, a, inca, y, incy);
vmzAbs( n, a, y, mode );
vmzAbsI(n, a, inca, y, incy, mode);

```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsAbs</code> , <code>vmsAbs</code> <code>const double*</code> for <code>vdAbs</code> , <code>vmdAbs</code> <code>const MKL_Complex8*</code> for <code>vcAbs</code> , <code>vmcAbs</code> <code>const MKL_Complex16*</code> for <code>vzAbs</code> , <code>vmzAbs</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca</i> , <i>incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsAbs</code> , <code>vmsAbs</code> , <code>vcAbs</code> , <code>vmcAbs</code> <code>double*</code> for <code>vdAbs</code> , <code>vmdAbs</code> , <code>vzAbs</code> , <code>vmzAbs</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Abs` function computes an absolute value of vector elements.

Special Values for Real Function v?Abs(x)

Argument	Result	Exception
+0	+0	
-0	+0	
$+\infty$	$+\infty$	
$-\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	INVALID

Specifications for special values of the complex functions are defined according to the following formula

$\text{Abs}(z) = \text{Hypot}(\text{RE}(z), \text{IM}(z))$.

v?Arg

Computes argument of vector elements.

Syntax

```
vcArg( n, a, y );
vcArgI(n, a, inca, y, incy);
vmcArg( n, a, y, mode );
vmcArgI(n, a, inca, y, incy, mode);
vzArg( n, a, y );
vzArgI(n, a, inca, y, incy);
vmzArg( n, a, y, mode );
vmzArgI(n, a, inca, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const MKL_Complex8* for vcArg, vmcArg const MKL_Complex16* for vzArg, vmcArg	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	float* for vcArg, vmcArg double* for vzArg, vmcArg	Pointer to an array that contains the output vector y .

Description

The $v?Arg$ function computes argument of vector elements.

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function $v?Arg(z)$

$RE(z)$ $i \cdot IM(z)$	$-\infty$	$-X$	-0	$+0$	$+X$	$+\infty$	NAN
$+i \cdot \infty$	$+3 \cdot \pi/4$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/4$	NAN
$+i \cdot Y$	$+\pi$		$+\pi/2$	$+\pi/2$		$+0$	NAN
$+i \cdot 0$	$+\pi$	$+\pi$	$+\pi$	$+0$	$+0$	$+0$	NAN
$-i \cdot 0$	$-\pi$	$-\pi$	$-\pi$	-0	-0	-0	NAN
$-i \cdot Y$	$-\pi$		$-\pi/2$	$-\pi/2$		-0	NAN
$-i \cdot \infty$	$-3 \cdot \pi/4$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/4$	NAN
$+i \cdot \text{NAN}$	NAN	NAN	NAN	NAN	NAN	NAN	NAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- $Arg(z) = \text{Atan2}(IM(z), RE(z))$.

$v?LinearFrac$

Performs linear fraction transformation of vectors a and b with scalar parameters.

Syntax

```
vsLinearFrac( n, a, b, scalea, shifta, scaleb, shiftb, y );
vsLinearFracI( n, a, inca, b, incb, scalea, shifta, scaleb, shiftb, y, incy );
vmsLinearFrac( n, a, b, scalea, shifta, scaleb, shiftb, y, mode );
vmsLinearFracI( n, a, inca, b, incb, scalea, shifta, scaleb, shiftb, y, incy, mode );
vdLinearFrac( n, a, b, scalea, shifta, scaleb, shiftb, y );
vdLinearFracI( n, a, inca, b, incb, scalea, shifta, scaleb, shiftb, y, incy );
vmdLinearFrac( n, a, b, scalea, shifta, scaleb, shiftb, y, mode );
vmdLinearFracI( n, a, inca, b, incb, scalea, shifta, scaleb, shiftb, y, incy, mode );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	const float* for vsLinearFrac, vmsLinearFrac const double* for vdLinearFrac, vmdLinearFrac	Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>scalea, scaleb</i>	const float for vsLinearFrac, vmsLinearFrac const double for vdLinearFrac, vmdLinearFrac	Constant values for scaling multipliers of vectors <i>a</i> and <i>b</i> .
<i>shifta, shiftb</i>	const float for vsLinearFrac, vmsLinearFrac const double for vdLinearFrac, vmdLinearFrac	Constant values for shifting addends of vectors <i>a</i> and <i>b</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsLinearFrac, vmsLinearFrac double* for vdLinearFrac, vmdLinearFrac	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?LinearFrac` function performs a linear fraction transformation of vector *a* by vector *b* with scalar parameters: scaling multipliers *scalea*, *scaleb* and shifting addends *shifta*, *shiftb*:

$$y[i] = (scalea \cdot a[i] + shifta) / (scaleb \cdot b[i] + shiftb), i = 1, 2 \dots n$$

The `v?LinearFrac` function is implemented in the EP accuracy mode only, therefore no special values are defined for this function. If used in HA or LA mode, `v?LinearFrac` sets the VM Error Status to `VML_STATUS_ACCURACYWARNING` (see the [Values of the VM Status](#) table). Correctness is guaranteed within the threshold limitations defined for each input parameter (see the table below); otherwise, the behavior is unspecified.

Threshold Limitations on Input Parameters

$$2^{E_{MIN}/2} \leq |scalea| \leq 2^{(E_{MAX}-2)/2}$$

Threshold Limitations on Input Parameters

$$2^{E_{\text{MIN}}/2} \leq |scaleb| \leq 2^{(E_{\text{MAX}}-2)/2}$$

$$|shiffta| \leq 2^{E_{\text{MAX}}-2}$$

$$|shifftb| \leq 2^{E_{\text{MAX}}-2}$$

$$2^{E_{\text{MIN}}/2} \leq a[i] \leq 2^{(E_{\text{MAX}}-2)/2}$$

$$2^{E_{\text{MIN}}/2} \leq b[i] \leq 2^{(E_{\text{MAX}}-2)/2}$$

$$a[i] \neq - (shiffta/scalea) * (1-\delta_1), \quad |\delta_1| \leq 2^{1-(p-1)/2}$$

$$b[i] \neq - (shifftb/scaleb) * (1-\delta_2), \quad |\delta_2| \leq 2^{1-(p-1)/2}$$

E_{MIN} and E_{MAX} are the minimum and maximum exponents and p is the number of significant bits (precision) for the corresponding data type according to the ANSI/IEEE Standard 754-2008 ([IEEE754]):

- for single precision $E_{\text{MIN}} = -126$, $E_{\text{MAX}} = 127$, $p = 24$
- for double precision $E_{\text{MIN}} = -1022$, $E_{\text{MAX}} = 1023$, $p = 53$

The thresholds become less strict for common cases with $scalea=0$ and/or $scaleb=0$:

- if $scalea=0$, there are no limitations for the values of $a[i]$ and $shiffta$.
- if $scaleb=0$, there are no limitations for the values of $b[i]$ and $shifftb$.

Example

To use the `vdLinearFrac` to shift vector a by a scalar value, set $scaleb$ to 0. Note that even if $scaleb$ is 0, b must be declared.

```
#include <stdio.h>
#include "mkl_vml.h"

int main()
{
    double a[10], *b;
    double r[10];
    double scalea = 1.0, scaleb = 0.0;
    double shiffta = -1.0, shifftb = 1.0;

    MKL_INT i=0,n=10;

    a[0]=-10000.0000;
    a[1]=-7777.7777;
    a[2]=-5555.5555;
    a[3]=-3333.3333;
    a[4]=-1111.1111;
    a[5]=1111.1111;
    a[6]=3333.3333;
    a[7]=5555.5555;
    a[8]=7777.7777;
    a[9]=10000.0000;

    vdLinearFrac( n, a, b, scalea, shiffta, scaleb, shifftb, r );

    for(i=0;i<10;i++) {
        printf("%25.14f %25.14f\n",a[i],r[i]);
    }
}
```

```
    return 0;
}
```

To use the `v?LinearFrac` to compute $shifta/(scaleb \cdot b[i] + shiftb)$, set `scalea` to 0. Note that even if `scalea` is 0, `a` must be declared.

v?Fmod

The `v?Fmod` function performs element by element computation of the modulus function of vector *a* with respect to vector *b*.

Syntax

```
vsFmod (n, a, b, y);
vsFmodI(n, a, inca, b, incb, y, incy);
vmsFmod (n, a, b, y, mode);
vmsFmodI(n, a, inca, b, incb, y, incy, mode);
vdFmod (n, a, b, y);
vdFmodI(n, a, inca, b, incb, y, incy);
vmdFmod (n, a, b, y, mode);
vmdFmodI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a, b</i>	<code>const float*</code> for <code>vsFmod</code> <code>const float*</code> for <code>vmsFmod</code> <code>const double*</code> for <code>vdFmod</code> <code>const double*</code> for <code>vmdFmod</code>	Pointers to arrays containing the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	float* for vsFmod	Pointer to an array containing the output vector y .
	float* for vmsFmod	
	double* for vdFmod	
	double* for vmdFmod	

Description

The `v?Fmod` function computes the modulus function of each element of vector a , with respect to the corresponding elements of vector b :

$$a_i - b_i * \text{trunc}(a_i / b_i)$$

In general, the modulus function `fmod(a_i , b_i)` returns the value $a_i - n * b_i$ for some integer n such that if b_i is nonzero, the result has the same sign as a_i and a magnitude less than the magnitude of b_i .

Special values for Real Function `v?Fmod(x, y)`

Argument 1	Argument 2	Result	VM Error Status	Exception
x not NAN	± 0	NAN	VML_STATUS_SING	INVALID
$\pm \infty$	y not NAN	NAN	VML_STATUS_SING	INVALID
± 0	$y \neq 0$, not NAN	± 0		
x finite	$\pm \infty$	x		UNDERFLOW if x is subnormal
NAN	y	NAN		
x	NAN	NAN		

NOTE

If element i in the result of `v?Fmod` is 0, its sign is that of a_i .

See Also

[Div](#) Performs element by element division of vector a by vector b

[Remainder](#) Performs element by element computation of the remainder function on the elements of vector a and the corresponding elements of vector b .

`v?Remainder`

Performs element by element computation of the remainder function on the elements of vector a and the corresponding elements of vector b .

Syntax

```
vsRemainder (n, a, b, y);
vsRemainderI(n, a, inca, b, incb, y, incy);
vmsRemainder (n, a, b, y, mode);
vmsRemainderI(n, a, inca, b, incb, y, incy, mode);
vdRemainder (n, a, b, y);
vdRemainderI(n, a, inca, b, incb, y, incy);
```

```
vmdRemainder (n, a, b, y, mode);
vmdRemainderI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	const float* for vsRemainder const float* for vmsRemainder const double* for vdRemainder const double* for vmdRemainder	Pointers to arrays containing the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsRemainder float* for vmsRemainder double* for vdRemainder double* for vmdRemainder	Pointer to an array containing the output vector <i>y</i> .

Description

Computes the remainder of each element of vector *a*, with respect to the corresponding elements of vector *b*: compute the values of *n* such that

$$n = a_i - n * b_i$$

where *n* is the integer nearest to the exact value of a_i/b_i . If two integers are equally close to a_i/b_i , *n* is the even one. If *n* is zero, it has the same sign as a_i .

Special values for Real Function v?Remainder(x, y)

Argument 1	Argument 2	Result	VM Error Status	Exception
x not NAN	±0	NAN	VML_STATUS_DOM	INVALID
±∞	y not NAN	NAN		INVALID
±0	y ≠ 0, not NAN	±0		
x finite	±∞	x		UNDERFLOW if x is subnormal
NAN	y	NAN		

Argument 1	Argument 2	Result	VM Error Status	Exception
x	NAN	NAN		

NOTE

If element i in the result of `v?Remainder` is 0, its sign is that of a_i .

See Also

Div Performs element by element division of vector a by vector b

Fmod The `v?Fmod` function performs element by element computation of the modulus function of vector a with respect to vector b .

Power and Root Functions**v?Inv**

Performs element by element inversion of the vector.

Syntax

```
vsInv( n, a, y );
vsInvI(n, a, inca, y, incy);
vmsInv( n, a, y, mode );
vmsInvI(n, a, inca, y, incy, mode);
vdInv( n, a, y );
vdInvI(n, a, inca, y, incy);
vmdInv( n, a, y, mode );
vmdInvI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
n	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
a	<code>const float*</code> for <code>vsInv</code> , <code>vmsInv</code> <code>const double*</code> for <code>vdInv</code> , <code>vmdInv</code>	Pointer to an array that contains the input vector a .
$inca, incy$	<code>const MKL_INT</code>	Specifies increments for the elements of a and y .
$mode$	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	float* for vsInv, vmsInv double* for vdInv, vmdInv	Pointer to an array that contains the output vector y .

Description

The $v?Inv$ function performs element by element inversion of the vector.

Special Values for Real Function $v?Inv(x)$

Argument	Result	VM Error Status	Exception
+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$+\infty$	+0		
$-\infty$	-0		
QNAN	QNAN		
SNAN	QNAN		INVALID

$v?Div$

Performs element by element division of vector a by vector b

Syntax

```
vsDiv( n, a, b, y );
vsDivI(n, a, inca, b, incb, y, incy);
vmsDiv( n, a, b, y, mode );
vmsDivI(n, a, inca, b, incb, y, incy, mode);
vdDiv( n, a, b, y );
vdDivI(n, a, inca, b, incb, y, incy);
vmdDiv( n, a, b, y, mode );
vmdDivI(n, a, inca, b, incb, y, incy, mode);
vcDiv( n, a, b, y );
vcDivI(n, a, inca, b, incb, y, incy);
vmcDiv( n, a, b, y, mode );
vmcDivI(n, a, inca, b, incb, y, incy, mode);
vzDiv( n, a, b, y );
vzDivI(n, a, inca, b, incb, y, incy);
vmzDiv( n, a, b, y, mode );
vmzDivI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	const float* for vsDiv, vmsDiv const double* for vdDiv, vmdDiv const MKL_Complex8* for vcDiv, vmcDiv const MKL_Complex16* for vzDiv, vmzDiv	Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Div Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < \text{abs}(b[i]) * \text{FLT_MAX}$
double precision	$\text{abs}(a[i]) < \text{abs}(b[i]) * \text{DBL_MAX}$

Precision overflow thresholds for the complex v?Div function are beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsDiv, vmsDiv double* for vdDiv, vmdDiv MKL_Complex8* for vcDiv, vmcDiv MKL_Complex16* for vzDiv, vmzDiv	Pointer to an array that contains the output vector <i>y</i> .

Description

The v?Div function performs element by element division of vector *a* by vector *b*.

Special values for Real Function v?Div(x)

Argument 1	Argument 2	Result	VM Error Status	Exception
X > +0	+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
X > +0	-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
X < +0	+0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
X < +0	-0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
+0	+0	QNAN	VML_STATUS_SING	
-0	-0	QNAN	VML_STATUS_SING	
X > +0	$+\infty$	+0		
X > +0	$-\infty$	-0		

Argument 1	Argument 2	Result	VM Error Status	Exception
$+\infty$	$+\infty$	QNAN		
$-\infty$	$-\infty$	QNAN		
QNAN	QNAN	QNAN		
SNAN	SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Div}(x1+i*y1, x2+i*y2) = (x1+i*y1) * (x2-i*y2) / (x2*x2+y2*y2).$$

Overflow in a complex function occurs when $x2+i*y2$ is not zero, $x1, x2, y1, y2$ are finite numbers, but the real or imaginary part of the exact result is so large that it does not fit the target precision. In that case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW`.

v?Sqrt

Computes a square root of vector elements.

Syntax

```
vsSqrt( n, a, y );
vsSqrtI(n, a, inca, y, incy);
vmsSqrt( n, a, y, mode );
vmsSqrtI(n, a, inca, y, incy, mode);
vdSqrt( n, a, y );
vdSqrtI(n, a, inca, y, incy);
vmdSqrt( n, a, y, mode );
vmdSqrtI(n, a, inca, y, incy, mode);
vcSqrt( n, a, y );
vcSqrtI(n, a, inca, y, incy);
vmcSqrt( n, a, y, mode );
vmcSqrtI(n, a, inca, y, incy, mode);
vzSqrt( n, a, y );
vzSqrtI(n, a, inca, y, incy);
vmzSqrt( n, a, y, mode );
vmzSqrtI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a</i>	const float* for vsSqrt, vmsSqrt const double* for vdSqrt, vmdSqrt const MKL_Complex8* for vcSqrt, vmcSqrt const MKL_Complex16* for vzSqrt, vmzSqrt	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsSqrt, vmsSqrt double* for vdSqrt, vmdSqrt MKL_Complex8* for vcSqrt, vmcSqrt MKL_Complex16* for vzSqrt, vmzSqrt	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Sqrt` function computes a square root of vector elements.

Special Values for Real Function `v?Sqrt(x)`

Argument	Result	VM Error Status	Exception
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
$+0$	$+0$		
-0	-0		
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function `v?Sqrt(z)`

RE(z) i·IM(z)	$-\infty$	$-X$	-0	$+0$	$+X$	$+\infty$	NAN
$+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$	$+\infty+i\cdot\infty$
$+i\cdot Y$	$+0+i\cdot\infty$					$+\infty+i\cdot 0$	QNAN+i·QNAN
$+i\cdot 0$	$+0+i\cdot\infty$		$+0+i\cdot 0$	$+0+i\cdot 0$		$+\infty+i\cdot 0$	QNAN+i·QNAN
$-i\cdot 0$	$+0-i\cdot\infty$		$+0-i\cdot 0$	$+0-i\cdot 0$		$+\infty-i\cdot 0$	QNAN+i·QNAN

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
-i·Y	+0-i·∞					+∞-i·0	QNAN+i·QNAN
-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞	+∞-i·∞
+i·NAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- raises `INVALID` exception when the real or imaginary part of the argument is `SNAN`
- `Sqrt(CONJ(z))=CONJ(Sqrt(z))`.

v?InvSqrt

Computes an inverse square root of vector elements.

Syntax

```
vsInvSqrt( n, a, y );
vsInvSqrtI(n, a, inca, y, incy);
vmsInvSqrt( n, a, y, mode );
vmsInvSqrtI(n, a, inca, y, incy, mode);
vdInvSqrt( n, a, y );
vdInvSqrtI(n, a, inca, y, incy);
vmdInvSqrt( n, a, y, mode );
vmdInvSqrtI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>n</code>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<code>a</code>	<code>const float*</code> for <code>vsInvSqrt</code> , <code>vmsInvSqrt</code> <code>const double*</code> for <code>vdInvSqrt</code> , <code>vmdInvSqrt</code>	Pointer to an array that contains the input vector <code>a</code> .
<code>inca, incy</code>	<code>const MKL_INT</code>	Specifies increments for the elements of <code>a</code> and <code>y</code> .
<code>mode</code>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsInvSqrt, vmsInvSqrt double* for vdInvSqrt, vmdInvSqrt	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?InvSqrt` function computes an inverse square root of vector elements.

Special Values for Real Function `v?InvSqrt(x)`

Argument	Result	VM Error Status	Exception
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
$+0$	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+0$		
QNAN	QNAN		
SNAN	QNAN		INVALID

`v?Cbrt`

Computes a cube root of vector elements.

Syntax

```
vsCbrt( n, a, y );
vsCbrtI(n, a, inca, y, incy);
vmsCbrt( n, a, y, mode );
vmsCbrtI(n, a, inca, y, incy, mode);
vdCbrt( n, a, y );
vdCbrtI(n, a, inca, y, incy);
vmdCbrt( n, a, y, mode );
vmdCbrtI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsCbrt, vmsCbrt const double* for vdCbrt, vmdCbrt	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .

Name	Type	Description
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsCbrt, vmsCbrt double* for vdCbrt, vmdCbrt	Pointer to an array that contains the output vector <i>y</i> .

Description

The *v?Cbrt* function computes a cube root of vector elements.

Special Values for Real Function *v?Cbrt(x)*

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
QNAN	QNAN	
SNAN	QNAN	INVALID

v?InvCbrt

Computes an inverse cube root of vector elements.

Syntax

```
vsInvCbrt( n, a, y );
vsInvCbrtI(n, a, inca, y, incy);
vmsInvCbrt( n, a, y, mode );
vmsInvCbrtI(n, a, inca, y, incy, mode);
vdInvCbrt( n, a, y );
vdInvCbrtI(n, a, inca, y, incy);
vmdInvCbrt( n, a, y, mode );
vmdInvCbrtI(n, a, inca, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a</i>	const float* for vsInvCbrt, vmsInvCbrt const double* for vdInvCbrt, vmdInvCbrt	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsInvCbrt, vmsInvCbrt double* for vdInvCbrt, vmdInvCbrt	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?InvCbrt` function computes an inverse cube root of vector elements.

Special Values for Real Function `v?InvCbrt(x)`

Argument	Result	VM Error Status	Exception
+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$+\infty$	+0		
$-\infty$	-0		
QNaN	QNaN		
SNAN	QNaN		INVALID

`v?Pow2o3`

Computes the cube root of the square of each vector element.

Syntax

```
vsPow2o3( n, a, y );
vsPow2o3I( n, a, inca, y, incy );
vmsPow2o3( n, a, y, mode );
vmsPow2o3I( n, a, inca, y, incy, mode );
vdPow2o3( n, a, y );
vdPow2o3I( n, a, inca, y, incy );
vmdPow2o3( n, a, y, mode );
vmdPow2o3I( n, a, inca, y, incy, mode );
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsPow2o3, vmsPow2o3 const double* for vdPow2o3, vmdPow2o3	Pointers to arrays that contain the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsPow2o3, vmsPow2o3 double* for vdPow2o3, vmdPow2o3	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Pow2o3` function computes the cube root of the square of each vector element.

Special Values for Real Function `v?Pow2o3(x)`

Argument	Result	Exception
+0	+0	
-0	+0	
$+\infty$	$+\infty$	
$-\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	INVALID

`v?Pow3o2`

Computes the square root of the cube of each vector element.

Syntax

```
vsPow3o2( n, a, y );
vsPow3o2I(n, a, inca, y, incy);
vmsPow3o2( n, a, y, mode );
vmsPow3o2I(n, a, inca, y, incy, mode);
vdPow3o2( n, a, y );
vdPow3o2I(n, a, inca, y, incy);
vmdPow3o2( n, a, y, mode );
```

```
vmdPow3o2I(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsPow3o2</code> , <code>vmsPow3o2</code> <code>const double*</code> for <code>vdPow3o2</code> , <code>vmdPow3o2</code>	Pointers to arrays that contain the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Pow3o2 Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT_MAX})^{2/3}$
double precision	$\text{abs}(a[i]) < (\text{DBL_MAX})^{2/3}$

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsPow3o2</code> , <code>vmsPow3o2</code> <code>double*</code> for <code>vdPow3o2</code> , <code>vmdPow3o2</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Pow3o2` function computes the square root of the cube of each vector element.

Special Values for Real Function v?Pow3o2(x)

Argument	Result	VM Error Status	Exception
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
$+0$	$+0$		
-0	-0		
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Pow

Computes *a* to the power *b* for elements of two vectors.

Syntax

```
vsPow( n, a, b, y );
vsPowI(n, a, inca, b, incb, y, incy);
vmsPow( n, a, b, y, mode );
vmsPowI(n, a, inca, b, incb, y, incy, mode);
vdPow( n, a, b, y );
vdPowI(n, a, inca, b, incb, y, incy);
vmdPow( n, a, b, y, mode );
vmdPowI(n, a, inca, b, incb, y, incy, mode);
vcPow( n, a, b, y );
vcPowI(n, a, inca, b, incb, y, incy);
vmcPow( n, a, b, y, mode );
vmcPowI(n, a, inca, b, incb, y, incy, mode);
vzPow( n, a, b, y );
vzPowI(n, a, inca, b, incb, y, incy);
vmzPow( n, a, b, y, mode );
vmzPowI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	const float* for vsPow, vmsPow const double* for vdPow, vmdPow const MKL_Complex8* for vcPow, vmcPow const MKL_Complex16* for vzPow, vmzPow	Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Pow Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT_MAX})^{1/b[i]}$

Data Type	Threshold Limitations on Input Parameters
double precision	$\text{abs}(a[i]) < (\text{DBL_MAX})^{1/b[i]}$

Precision overflow thresholds for the complex `v?Pow` function are beyond the scope of this document.

Output Parameters

Name	Type	Description
<code>y</code>	<code>float*</code> for <code>vsPow</code> , <code>vmsPow</code> <code>double*</code> for <code>vdPow</code> , <code>vmdPow</code> <code>MKL_Complex8*</code> for <code>vcPow</code> , <code>vmcPow</code> <code>MKL_Complex16*</code> for <code>vzPow</code> , <code>vmzPow</code>	Pointer to an array that contains the output vector <code>y</code> .

Description

The `v?Pow` function computes a to the power b for elements of two vectors.

The real function `v(s/d)Pow` has certain limitations on the input range of a and b parameters. Specifically, if $a[i]$ is positive, then $b[i]$ may be arbitrary. For negative $a[i]$, the value of $b[i]$ must be an integer (either positive or negative).

The complex function `v(c/z)Pow` has no input range limitations.

Special values for Real Function `v?Pow(x,y)`

Argument 1 (X)	Argument 2 (Y)	Result	VM Error Status	Exception
+0	neg. odd integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. odd integer	$-\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
+0	neg. even integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. even integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
+0	neg. non-integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. non-integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	pos. odd integer	+0		
-0	pos. odd integer	-0		
+0	pos. even integer	+0		
-0	pos. even integer	+0		
+0	pos. non-integer	+0		
-0	pos. non-integer	+0		
-1	$+\infty$	+1		
-1	$-\infty$	+1		
+1	any value	+1		
+1	+0	+1		
+1	-0	+1		
+1	$+\infty$	+1		
+1	$-\infty$	+1		
+1	QNaN	+1		
any value	+0	+1		
+0	+0	+1		
-0	+0	+1		

Argument 1 (X)	Argument 2 (Y)	Result	VM Error Status	Exception
$+\infty$	+0	+1		
$-\infty$	+0	+1		
QNAN	+0	+1		
any value	-0	+1		
+0	-0	+1		
-0	-0	+1		
$+\infty$	-0	+1		
$-\infty$	-0	+1		
QNAN	-0	+1		
$X < +0$	non-integer	QNAN	VML_STATUS_ERRDOM	INVALID
$ X < 1$	$-\infty$	$+\infty$		
+0	$-\infty$	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	$-\infty$	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
$ X > 1$	$-\infty$	+0		
$+\infty$	$-\infty$	+0		
$-\infty$	$-\infty$	+0		
$ X < 1$	$+\infty$	+0		
+0	$+\infty$	+0		
-0	$+\infty$	+0		
$ X > 1$	$+\infty$	$+\infty$		
$+\infty$	$+\infty$	$+\infty$		
$-\infty$	$+\infty$	$+\infty$		
$-\infty$	neg. odd integer	-0		
$-\infty$	neg. even integer	+0		
$-\infty$	neg. non-integer	+0		
$-\infty$	pos. odd integer	$-\infty$		
$-\infty$	pos. even integer	$+\infty$		
$-\infty$	pos. non-integer	$+\infty$		
$+\infty$	$X < +0$	+0		
$+\infty$	$X > +0$	$+\infty$		
Big finite value*	Big finite value*	$+/-\infty$	VML_STATUS_OVERFLOW	OVERFLOW
QNAN	QNAN	QNAN		
QNAN	SNAN	QNAN		INVALID
SNAN	QNAN	QNAN		INVALID
SNAN	SNAN	QNAN		INVALID

The complex double precision versions of this function, `vzPow` and `vmzPow`, are implemented in the EP accuracy mode only. If used in HA or LA mode, `vzPow` and `vmzPow` set the VM Error Status to `VML_STATUS_ACCURACYWARNING` (see the [Values of the VM Status table](#)).

* Overflow in a real function is supported only in the HA/LA accuracy modes. The overflow occurs when x and y are finite numbers, but the result is too large to fit the target precision. In this case, the function:

1. Returns ∞ in the result.
2. Raises the `OVERFLOW` exception.
3. Sets the VM Error Status to `VML_STATUS_OVERFLOW`.

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all $\text{RE}(x)$, $\text{RE}(y)$, $\text{IM}(x)$, $\text{IM}(y)$ arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

v?Powx

Computes vector a to the scalar power b .

Syntax

```
vsPowx( n, a, b, y );
vsPowxI(n, a, inca, b, y, incy);
vmsPowx( n, a, b, y, mode );
vmsPowxI(n, a, inca, b, y, incy, mode);
vdPowx( n, a, b, y );
vdPowxI(n, a, inca, b, y, incy);
vmdPowx( n, a, b, y, mode );
vmdPowxI(n, a, inca, b, y, incy, mode);
vcPowx( n, a, b, y );
vcPowxI(n, a, inca, b, y, incy);
vmcPowx( n, a, b, y, mode );
vmcPowxI(n, a, inca, b, y, incy, mode);
vzPowx( n, a, b, y );
vzPowxI(n, a, inca, b, y, incy);
vmzPowx( n, a, b, y, mode );
vmzPowxI(n, a, inca, b, y, incy, mode);
```

Include Files

- `mkh.h`

Input Parameters

Name	Type	Description
n	<code>const MKL_INT</code>	Number of elements to be calculated.
a	<code>const float*</code> for <code>vsPowx</code> , <code>vmsPowx</code> <code>const double*</code> for <code>vdPowx</code> , <code>vmdPowx</code> <code>const MKL_Complex8*</code> for <code>vcPowx</code> , <code>vmcPowx</code> <code>const MKL_Complex16*</code> for <code>vzPowx</code> , <code>vmzPowx</code>	Pointer to an array that contains the input vector a .
b	<code>const float</code> for <code>vsPowx</code> , <code>vmsPowx</code>	Constant value for power b .

Name	Type	Description
	const double for vdPowx, vmdPowx	
	const MKL_Complex8 for vcPowx, vmcPowx	
	const MKL_Complex16 for vzPowx, vmzPowx	
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Powx Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT_MAX})^{1/b}$
double precision	$\text{abs}(a[i]) < (\text{DBL_MAX})^{1/b}$

Precision overflow thresholds for the complex v?Powx function are beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsPowx, vmsPowx double* for vdPowx, vmdPowx MKL_Complex8* for vcPowx, vmcPowx MKL_Complex16* for vzPowx, vmzPowx	Pointer to an array that contains the output vector <i>y</i> .

Description

The v?Powx function computes *a* to the power *b* for a vector *a* and a scalar *b*.

The real function v(s/d)Powx has certain limitations on the input range of *a* and *b* parameters. Specifically, if *a*[*i*] is positive, then *b* may be arbitrary. For negative *a*[*i*], the value of *b* must be an integer (either positive or negative).

The complex function v(c/z)Powx has no input range limitations.

Special values and VM Error Status treatment are the same as for the v?Pow function.

v?Powr

*Computes *a* to the power *b* for elements of two vectors, where the elements of vector argument *a* are all non-negative.*

Syntax

```
vsPowr (n, a, b, y);
vsPowrI(n, a, inca, b, incb, y, incy);
vmsPowr (n, a, b, y, mode);
vmsPowrI(n, a, inca, b, incb, y, incy, mode);
```



```

vdPowr (n, a, b, y);
vdPowrI(n, a, inca, b, incb, y, incy);
vmdPowr (n, a, b, y, mode);
vmdPowrI(n, a, inca, b, incb, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	const float* for vsPowr const float* for vmsPowr const double* for vdPowr const double* for vmdPowr	Pointers to arrays containing the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsPowr float* for vmsPowr double* for vdPowr double* for vmdPowr	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Powr` function raises each element of vector *a* by the corresponding element of vector *b*. The elements of *a* are all nonnegative ($a_i \geq 0$).

Precision Overflow Thresholds for Real Function v?Powr

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < (\text{FLT_MAX})^{1/b_i}$
double precision	$a_i < (\text{DBL_MAX})^{1/b_i}$

Special values and VM Error Status treatment for `v?Powr` function are the same as for `v?Pow`, unless otherwise indicated in this table:

Special values for Real Function v?Powr(x)

Argument 1	Argument 2	Result	VM Error Status	Exception
$x < 0$	any value <i>y</i>	NAN	VML_STATUS_ERRDOM	INVALID

Argument 1	Argument 2	Result	VM Error Status	Exception
$0 < x < \infty$	± 0	1		
± 0	$-\infty < y < 0$	$+\infty$		
± 0	$-\infty$	$+\infty$		
± 0	$y > 0$	$+0$		
1	$-\infty < y < \infty$	1		
± 0	± 0	NAN		
$+\infty$	± 0	NAN		
1	$+\infty$	NAN		
$x \geq 0$	NAN	NAN		
NAN	any value y	NAN		
$0 < x < 1$	$-\infty$	$+\infty$		
$x > 1$	$-\infty$	$+0$		
$0 \leq x < 1$	$+\infty$	$+0$		
$x > 1$	$+\infty$	$+\infty$		
$+\infty$	$x < +0$	$+0$		
$+\infty$	$x > +0$	$+\infty$		
QNAN	QNAN	QNAN	VML_STATUS_ERRDOM	
QNAN	SNAN	QNAN	VML_STATUS_ERRDOM	INVALID
SNAN	QNAN	QNAN	VML_STATUS_ERRDOM	INVALID
SNAN	SNAN	QNAN	VML_STATUS_ERRDOM	INVALID

See Also

[Pow](#) Computes a to the power b for elements of two vectors.

[Powx](#) Computes vector a to the scalar power b .

v?Hypot

Computes a square root of sum of two squared elements.

Syntax

```
vsHypot( n, a, b, y );
vsHypotI(n, a, inca, b, incb, y, incy);
vmsHypot( n, a, b, y, mode );
vmsHypotI(n, a, inca, b, incb, y, incy, mode);
vdHypot( n, a, b, y );
vdHypotI(n, a, inca, b, incb, y, incy);
vmdHypot( n, a, b, y, mode );
vmdHypotI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Number of elements to be calculated.
<i>a, b</i>	const float* for vsHypot, vmsHypot const double* for vdHypot, vmdHypot	Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb,</i> <i>incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Precision Overflow Thresholds for Hypot Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < \text{sqrt}(\text{FLT_MAX})$ $\text{abs}(b[i]) < \text{sqrt}(\text{FLT_MAX})$
double precision	$\text{abs}(a[i]) < \text{sqrt}(\text{DBL_MAX})$ $\text{abs}(b[i]) < \text{sqrt}(\text{DBL_MAX})$

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsHypot, vmsHypot double* for vdHypot, vmdHypot	Pointer to an array that contains the output vector <i>y</i> .

Description

The function `v?Hypot` computes a square root of sum of two squared elements.

Special values for Real Function `v?Hypot(x)`

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
-0	-0	+0	
$+\infty$	any value	$+\infty$	
any value	$+\infty$	$+\infty$	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	any value	QNAN	
any value	QNAN	QNAN	

Exponential and Logarithmic Functions

v?Exp*Computes an exponential of vector elements.*

Syntax

```

vsExp( n, a, y );
vsExpI(n, a, inca, y, incy);
vmsExp( n, a, y, mode );
vmsExpI(n, a, inca, y, incy, mode);
vdExp( n, a, y );
vdExpI(n, a, inca, y, incy);
vmdExp( n, a, y, mode );
vmdExpI(n, a, inca, y, incy, mode);
vcExp( n, a, y );
vcExpI(n, a, inca, y, incy);
vmcExp( n, a, y, mode );
vmcExpI(n, a, inca, y, incy, mode);
vzExp( n, a, y );
vzExpI(n, a, inca, y, incy);
vmzExp( n, a, y, mode );
vmzExpI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsExp, vmsExp const double* for vdExp, vmdExp const MKL_Complex8* for vcExp, vmcExp const MKL_Complex16* for vzExp, vmzExp	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Exp Function

Data Type	Threshold Limitations on Input Parameters
single precision	$a[i] < \text{Ln}(\text{FLT_MAX})$
double precision	$a[i] < \text{Ln}(\text{DBL_MAX})$

Precision overflow thresholds for the complex v?Exp function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	float* for vsExp, vmsExp double* for vdExp, vmdExp MKL_Complex8* for vcExp, vmcExp MKL_Complex16* for vzExp, vmzExp	Pointer to an array that contains the output vector y .

Description

The v?Exp function computes an exponential of vector elements.

Special Values for Real Function v?Exp(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
$X > \text{overflow}$	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
$X < \text{underflow}$	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
$+\infty$	$+\infty$		
$-\infty$	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function v?Exp(z)

$\text{RE}(z)$ $i \cdot \text{IM}(z)$	$-\infty$	$-X$	-0	$+0$	$+X$	$+\infty$	NAN
$+i \cdot \infty$	$+0+i \cdot 0$	QNAN $+i \cdot \text{QNAN}$ INVALID	QNAN $+i \cdot \text{QNAN}$ INVALID	QNAN $+i \cdot \text{QNAN}$ INVALID	QNAN $+i \cdot \text{QNAN}$ INVALID	$+\infty$ $+i \cdot \text{QNAN}$ INVALID	QNAN $+i \cdot \text{QNAN}$ INVALID
$+i \cdot Y$	$+0 \cdot \text{CIS}(Y)$					$+\infty \cdot \text{CIS}(Y)$	QNAN $+i \cdot \text{QNAN}$
$+i \cdot 0$	$+0 \cdot \text{CIS}(0)$		$+1+i \cdot 0$	$+1+i \cdot 0$		$+\infty+i \cdot 0$	QNAN $+i \cdot 0$
$-i \cdot 0$	$+0 \cdot \text{CIS}(0)$		$+1-i \cdot 0$	$+1-i \cdot 0$		$+\infty-i \cdot 0$	QNAN $-i \cdot 0$
$-i \cdot Y$	$+0 \cdot \text{CIS}(Y)$					$+\infty \cdot \text{CIS}(Y)$	QNAN $+i \cdot \text{QNAN}$
$-i \cdot \infty$	$+0-i \cdot 0$	QNAN $+i \cdot \text{QNAN}$ INVALID	QNAN $+i \cdot \text{QNAN}$ INVALID	QNAN $+i \cdot \text{QNAN}$ INVALID	QNAN $+i \cdot \text{QNAN}$ INVALID	$+\infty$ $+i \cdot \text{QNAN}$ INVALID	QNAN $+i \cdot \text{QNAN}$ INVALID

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·NAN N	+0+i·0	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN	QNAN +i·QNAN

Notes:

- raises the `INVALID` exception when real or imaginary part of the argument is `SNAN`
- raises the `INVALID` exception on argument $z = -\infty + i \cdot \text{QNAN}$
- raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW` in the case of overflow, that is, when both `RE(z)` and `IM(z)` are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.

v?Exp2

Computes the base 2 exponential of vector elements.

Syntax

```
vsExp2 (n, a, y);
vsExp2I(n, a, inca, y, incy);
vmsExp2 (n, a, y, mode);
vmsExp2I(n, a, inca, y, incy, mode);
vdExp2 (n, a, y);
vdExp2I(n, a, inca, y, incy);
vmdExp2 (n, a, y, mode);
vmdExp2I(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsExp2</code> <code>const float*</code> for <code>vmsExp2</code> <code>const double*</code> for <code>vdExp2</code> <code>const double*</code> for <code>vmdExp2</code>	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	<code>float*</code> for <code>vsExp2</code>	Pointer to an array containing the output vector y .
	<code>float*</code> for <code>vmsExp2</code>	
	<code>double*</code> for <code>vdExp2</code>	
	<code>double*</code> for <code>vmdExp2</code>	

Description

The `v?Exp2` function computes the base 2 exponential of vector elements.

Precision Overflow Thresholds for Real Function `v?Exp2`

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < \log_2(\text{FLT_MAX})$
double precision	$a_i < \log_2(\text{DBL_MAX})$

See [Special Value Notations](#) for the conventions used in this table:

Special values for Real Function `v?Exp2(x)`

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
$x > \text{overflow}$	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
$x < \text{underflow}$	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
$+\infty$	$+\infty$		
$-\infty$	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[Exp](#) Computes an exponential of vector elements.

[Exp10](#) Computes the base 10 exponential of vector elements.

`v?Exp10`

Computes the base 10 exponential of vector elements.

Syntax

```
vsExp10 (n, a, y);
vsExp10I(n, a, inca, y, incy);
vmsExp10 (n, a, y, mode);
vmsExp10I(n, a, inca, y, incy, mode);
vdExp10 (n, a, y);
vdExp10I(n, a, inca, y, incy);
vmdExp10 (n, a, y, mode);
vmdExp10I(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsExp10</code> <code>const float*</code> for <code>vmsExp10</code> <code>const double*</code> for <code>vdExp10</code> <code>const double*</code> for <code>vmdExp10</code>	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsExp10</code> <code>float*</code> for <code>vmsExp10</code> <code>double*</code> for <code>vdExp10</code> <code>double*</code> for <code>vmdExp10</code>	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Exp10` function computes the base 10 exponential of vector elements.

Precision Overflow Thresholds for Real Function `v?Exp10`

Data Type	Threshold Limitations on Input Parameters
single precision	$a_i < \log_{10}(\text{FLT_MAX})$
double precision	$a_i < \log_{10}(\text{DBL_MAX})$

See [Special Value Notations](#) for the conventions used in this table:

Special values for Real Function `v?Pow(x)`

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
$x > \text{overflow}$	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
$x < \text{underflow}$	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
$+\infty$	$+\infty$		
$-\infty$	+0		
QNaN	QNaN		
SNAN	QNaN		INVALID

See Also

[Exp](#) Computes an exponential of vector elements.

[Exp2](#) Computes the base 2 exponential of vector elements.

v?Expm1

Computes an exponential of vector elements decreased by 1.

Syntax

```

vsExpm1( n, a, y );
vsExpm1I(n, a, inca, y, incy);
vmsExpm1( n, a, y, mode );
vmsExpm1I(n, a, inca, y, incy, mode);
vdExpm1( n, a, y );
vdExpm1I(n, a, inca, y, incy);
vmdExpm1( n, a, y, mode );
vmdExpm1I(n, a, inca, y, incy, mode);

```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsExpm1</code> , <code>vmsExpm1</code> <code>const double*</code> for <code>vdExpm1</code> , <code>vmdExpm1</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Expm1 Function

Data Type	Threshold Limitations on Input Parameters
single precision	$a[i] < \ln(\text{FLT_MAX})$
double precision	$a[i] < \ln(\text{DBL_MAX})$

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsExpm1</code> , <code>vmsExpm1</code> <code>double*</code> for <code>vdExpm1</code> , <code>vmdExpm1</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Expml` function computes an exponential of vector elements decreased by 1.

Special Values for Real Function `v?Expml(x)`

Argument	Result	VM Error Status	Exception
+0	+0		
-0	+0		
$X > \text{overflow}$	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
$+\infty$	$+\infty$		
$-\infty$	-1		
QNAN	QNAN		
SNAN	QNAN		INVALID

`v?Ln`

Computes natural logarithm of vector elements.

Syntax

```
vsLn( n, a, y );
vsLnI(n, a, inca, y, incy);
vmsLn( n, a, y, mode );
vmsLnI(n, a, inca, y, incy, mode);
vdLn( n, a, y );
vdLnI(n, a, inca, y, incy);
vmdLn( n, a, y, mode );
vmdLnI(n, a, inca, y, incy, mode);
vcLn( n, a, y );
vcLnI(n, a, inca, y, incy);
vmcLn( n, a, y, mode );
vmcLnI(n, a, inca, y, incy, mode);
vzLn( n, a, y );
vzLnI(n, a, inca, y, incy);
vmzLn( n, a, y, mode );
vmzLnI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>n</code>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a</i>	const float* for vsLn, vmsLn const double* for vdLn, vmdLn const MKL_Complex8* for vcLn, vmcLn const MKL_Complex16* for vzLn, vmzLn	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsLn, vmsLn double* for vdLn, vmdLn MKL_Complex8* for vcLn, vmcLn MKL_Complex16* for vzLn, vmzLn	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Ln` function computes natural logarithm of vector elements.

Special Values for Real Function `v?Ln(x)`

Argument	Result	VM Error Status	Exception
+1	+0		
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
+0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function `v?Ln(z)`

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+\infty$	$+\infty + i \cdot \frac{3\pi}{4}$	$+\infty + i\pi/2$	$+\infty + i\pi/2$	$+\infty + i\pi/2$	$+\infty + i\pi/2$	$+\infty + i\pi/4$	$+\infty + i\cdot\text{QNAN}$
$+i\cdot Y$	$+\infty + i\pi$					$+\infty + i\cdot 0$	QNAN $+i\cdot\text{QNAN}$

RE(z) i·IM(z))	-∞	-X	-0	+0	+X	+∞	NAN
							INVALID
+i·0	+∞+i·π		-∞+i·π ZERODIVID E	-∞+i·0 ZERODIVID E		+∞+i·0	QNAN +i·QNAN INVALID
-i·0	+∞-i·π		-∞-i·π ZERODIVID E	-∞-i·0 ZERODIVID E		+∞-i·0	QNAN +i·QNAN INVALID
-i·Y	+∞-i·π					+∞-i·0	QNAN +i·QNAN INVALID
-i·∞	+∞ - i · $\frac{3\pi}{4}$	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	+∞ +i·QNAN	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN	QNAN +i·QNAN INVALID

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`

v?Log2

Computes the base 2 logarithm of vector elements.

Syntax

```
vsLog2 (n, a, y);
vsLog2I(n, a, inca, y, incy);
vmsLog2 (n, a, y, mode);
vmsLog2I(n, a, inca, y, incy, mode);
vdLog2 (n, a, y);
vdLog2I(n, a, inca, y, incy);
vmdLog2 (n, a, y, mode);
vmdLog2I(n, a, inca, y, incy, mode);
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsLog2 const float* for vmsLog2 const double* for vdLog2 const double* for vmdLog2	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsLog2 float* for vmsLog2 double* for vdLog2 double* for vmdLog2	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Log2` function computes the base 2 logarithm of vector elements.

See [Special Value Notations](#) for the conventions used in this table:

Special values for Real Function `v?Log2(x)`

Argument	Result	VM Error Status	Exception
+1	+0		
$x < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
+0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[Ln](#) Computes natural logarithm of vector elements.

[Log10](#) Computes the base 10 logarithm of vector elements.

`v?Log10`

Computes the base 10 logarithm of vector elements.

Syntax

```

vsLog10( n, a, y );
vsLog10I(n, a, inca, y, incy);
vmsLog10( n, a, y, mode );
vmsLog10I(n, a, inca, y, incy, mode);
vdLog10( n, a, y );
vdLog10I(n, a, inca, y, incy);
vmdLog10( n, a, y, mode );
vmdLog10I(n, a, inca, y, incy, mode);
vcLog10( n, a, y );
vcLog10I(n, a, inca, y, incy);
vmcLog10( n, a, y, mode );
vmcLog10I(n, a, inca, y, incy, mode);
vzLog10( n, a, y );
vzLog10I(n, a, inca, y, incy);
vmzLog10( n, a, y, mode );
vmzLog10I(n, a, inca, y, incy, mode);

```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsLog10</code> , <code>vmsLog10</code> <code>const double*</code> for <code>vdLog10</code> , <code>vmdLog10</code> <code>const MKL_Complex8*</code> for <code>vcLog10</code> , <code>vmcLog10</code> <code>const MKL_Complex16*</code> for <code>vzLog10</code> , <code>vmzLog10</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	float* for vsLog10, vmsLog10 double* for vdLog10, vmdLog10 MKL_Complex8* for vcLog10, vmcLog10 MKL_Complex16* for vzLog10, vmzLog10	Pointer to an array that contains the output vector y.

Description

The v?Log10 function computes the base 10 logarithm of vector elements.

Special Values for Real Function v?Log10(x)

Argument	Result	VM Error Status	Exception
+1	+0		
X < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	-∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function v?Log10(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+	$+i \frac{3}{4} \frac{\pi}{\ln(10)}$	$+i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+i \frac{\pi}{4} \frac{1}{\ln(10)}$	$1 + i \cdot \text{QNAN}$ INVALID
+i·Y	$+i \frac{\pi}{\ln(10)}$					$+i \cdot 0$	QNAN +i·QNAN INVALID
+i·0	$+i \frac{\pi}{\ln(10)}$		$-i \frac{\pi}{\ln(10)}$ ZERODIVIDE E	$-i \frac{\pi}{\ln(10)}$ ZERODIVIDE		$+i \cdot 0$	QNAN +i·QNAN INVALID
-i·0	$-i \frac{\pi}{\ln(10)}$		$-i \frac{\pi}{\ln(10)}$ ZERODIVIDE E	$-i \frac{\pi}{\ln(10)}$ ZERODIVIDE		$+i \cdot 0$	QNAN- i·QNAN INVALID

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
$-i \cdot Y$ $+ \infty - i \frac{\pi}{\ln(10)}$						$+\infty - i \cdot 0$	QNAN +i·QNAN INVALID
$-i$ $+ \infty + i \frac{3}{4} \ln(10)$	$+ \infty - i \frac{\pi}{2} \ln(10)$	$+ \infty - i \frac{\pi}{2} \ln(10)$	$+ \infty - i \frac{\pi}{2} \ln(10)$	$+ \infty - i \frac{\pi}{2} \ln(10)$	$+ \infty - i \frac{\pi}{2} \ln(10)$	$+ \infty - i \frac{\pi}{4} \ln(10)$	$1 + \infty + i \cdot \text{QNAN}$
+i·NAN	$+\infty$ +i·QNAN	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	$+\infty$ +i·QNAN	QNAN +i·QNAN INVALID

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`

v?Log1p

Computes a natural logarithm of vector elements that are increased by 1.

Syntax

```
vsLog1p( n, a, y );
vsLog1pI(n, a, inca, y, incy);
vmsLog1p( n, a, y, mode );
vmsLog1pI(n, a, inca, y, incy, mode);
vdLog1p( n, a, y );
vdLog1pI(n, a, inca, y, incy);
vmdLog1p( n, a, y, mode );
vmdLog1pI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>n</code>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<code>a</code>	<code>const float*</code> for <code>vsLog1p</code> , <code>vmsLog1p</code> <code>const double*</code> for <code>vdLog1p</code> , <code>vmdLog1p</code>	Pointer to an array that contains the input vector <code>a</code> .

Name	Type	Description
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsLog1p, vmsLog1p double* for vdLog1p, vmdLog1p	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Log1p` function computes a natural logarithm of vector elements that are increased by 1.

Special Values for Real Function `v?Log1p(x)`

Argument	Result	VM Error Status	Exception
-1	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$X < -1$	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+0		
-0	-0		
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

`v?Logb`

*Computes the exponents of the elements of input vector *a*.*

Syntax

```
vsLogb (n, a, y);
vsLogbI(n, a, inca, y, incy);
vmsLogb (n, a, y, mode);
vmsLogbI(n, a, inca, y, incy, mode);
vdLogb (n, a, y);
vdLogbI(n, a, inca, y, incy);
vmdLogb (n, a, y, mode);
vmdLogbI(n, a, inca, y, incy, mode);
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsLogb const float* for vmsLogb const double* for vdLogb const double* for vmdLogb	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsLogb float* for vmsLogb double* for vdLogb double* for vmdLogb	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Logb` function computes the exponents of the elements of the input vector *a*. For each element a_i of vector *a*, this is the integral part of $\log_2|a_i|$. The returned value is exact and is independent of the current rounding direction mode.

See [Special Value Notations](#) for the conventions used in this table:

Special values for Real Function v?Logb(x)

Argument	Result	VM Error Status	Exception
+0	$-\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	$-\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
$-\infty$	$+\infty$		
$+\infty$	$+\infty$		
QNaN	QNaN		
SNAN	QNaN		INVALID

Trigonometric Functions

v?Cos

Computes cosine of vector elements.

Syntax

```
vsCos( n, a, y );
```

```

vsCosI(n, a, inca, y, incy);
vmsCos( n, a, y, mode );
vmsCosI(n, a, inca, y, incy, mode);
vdCos( n, a, y );
vdCosI(n, a, inca, y, incy);
vmdCos( n, a, y, mode );
vmdCosI(n, a, inca, y, incy, mode);
vcCos( n, a, y );
vcCosI(n, a, inca, y, incy);
vmcCos( n, a, y, mode );
vmcCosI(n, a, inca, y, incy, mode);
vzCos( n, a, y );
vzCosI(n, a, inca, y, incy);
vmzCos( n, a, y, mode );
vmzCosI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsCos, vmsCos const double* for vdCos, vmdCos const MKL_Complex8* for vcCos, vmcCos const MKL_Complex16* for vzCos, vmzCos	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsCos, vmsCos double* for vdCos, vmdCos MKL_Complex8* for vcCos, vmcCos	Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
------	------	-------------

MKL_Complex16* for vzCos, vmzCos

Description

The `v?Cos` function computes cosine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function `v?Cos(x)`

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Cos}(z) = \text{Cosh}(i * z).$$

Product and Performance Information

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

Notice revision #20201201

`v?Sin`

Computes sine of vector elements.

Syntax

```
vsSin( n, a, y );
vsSinI(n, a, inca, y, incy);
vmsSin( n, a, y, mode );
vmsSinI(n, a, inca, y, incy, mode);
vdSin( n, a, y );
vdSinI(n, a, inca, y, incy);
vmdSin( n, a, y, mode );
vmdSinI(n, a, inca, y, incy, mode);
vcSin( n, a, y );
vcSinI(n, a, inca, y, incy);
vmcSin( n, a, y, mode );
vmcSinI(n, a, inca, y, incy, mode);
```

```

vzSin( n, a, y );
vzSinI(n, a, inca, y, incy);
vmzSin( n, a, y, mode );
vmzSinI(n, a, inca, y, incy, mode);

```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsSin</code> , <code>vmsSin</code> <code>const double*</code> for <code>vdSin</code> , <code>vmdSin</code> <code>const MKL_Complex8*</code> for <code>vcSin</code> , <code>vmcSin</code> <code>const MKL_Complex16*</code> for <code>vzSin</code> , <code>vmzSin</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca</i> , <i>incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsSin</code> , <code>vmsSin</code> <code>double*</code> for <code>vdSin</code> , <code>vmdSin</code> <code>MKL_Complex8*</code> for <code>vcSin</code> , <code>vmcSin</code> <code>MKL_Complex16*</code> for <code>vzSin</code> , <code>vmzSin</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The function computes sine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?Sin(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID

Argument	Result	VM Error Status	Exception
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$\text{Sin}(z) = -i * \text{Sinh}(i * z).$

Product and Performance Information

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

Notice revision #20201201

v?SinCos

Computes sine and cosine of vector elements.

Syntax

```
vsSinCos( n, a, y, z );
vsSinCosI(n, a, inca, y, incy, z, incz);
vmsSinCos( n, a, y, z, mode );
vmsSinCosI(n, a, inca, y, incy, z, incz, mode);
vdSinCos( n, a, y, z );
vdSinCosI(n, a, inca, y, incy, z, incz);
vmdSinCos( n, a, y, z, mode );
vmdSinCosI(n, a, inca, y, incy, z, incz, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsSinCos, vmsSinCos const double* for vdSinCos, vmdSinCos	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy, incz</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>y</i> , and <i>z</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y, z	<code>float*</code> for <code>vsSinCos</code> , <code>vmsSinCos</code>	Pointers to arrays that contain the output vectors y (for sine values) and z (for cosine values).
	<code>double*</code> for <code>vdSinCos</code> , <code>vmdSinCos</code>	

Description

The function computes sine and cosine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function `v?SinCos(x)`

Argument	Result 1	Result 2	VM Error Status	Exception
+0	+0	+1		
-0	-0	+1		
$+\infty$	QNAN	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN	QNAN		
SNAN	QNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Sin}(z) = -i * \text{Sinh}(i * z).$$

Product and Performance Information

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

Notice revision #20201201

`v?CIS`

Computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Syntax

```
vcCIS( n, a, y );
vcCISI( n, a, inca, y, incy );
vmcCIS( n, a, y, mode );
vmcCISI( n, a, inca, y, incy, mode );
vzCIS( n, a, y );
vzCISI( n, a, inca, y, incy );
vmzCIS( n, a, y, mode );
vmzCISI( n, a, inca, y, incy, mode );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vcCIS</code> , <code>vmcCIS</code> <code>const double*</code> for <code>vzCIS</code> , <code>vmzCIS</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca</i> , <i>incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>MKL_Complex8*</code> for <code>vcCIS</code> , <code>vmcCIS</code> <code>MKL_Complex16*</code> for <code>vzCIS</code> , <code>vmzCIS</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?CIS` function computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function `v?CIS(x)`

x	CIS(x)
$+\infty$	QNAN+i·QNAN INVALID
$+0$	$+1+i\cdot 0$
-0	$+1-i\cdot 0$
$-\infty$	QNAN+i·QNAN INVALID
NAN	QNAN+i·QNAN

Notes:

- raises `INVALID` exception when the argument is `SNAN`
- raises `INVALID` exception and sets the VM Error Status to `VML_STATUS_ERRDOM` for $x=+\infty$, $x=-\infty$

`v?Tan`

Computes tangent of vector elements.

Syntax

```

vsTan( n, a, y );
vsTanI(n, a, inca, y, incy);
vmsTan( n, a, y, mode );
vmsTanI(n, a, inca, y, incy, mode);
vdTan( n, a, y );
vdTanI(n, a, inca, y, incy);
vmdTan( n, a, y, mode );
vmdTanI(n, a, inca, y, incy, mode);
vcTan( n, a, y );
vcTanI(n, a, inca, y, incy);
vmcTan( n, a, y, mode );
vmcTanI(n, a, inca, y, incy, mode);
vzTan( n, a, y );
vzTanI(n, a, inca, y, incy);
vmzTan( n, a, y, mode );
vmzTanI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsTan, vmsTan const double* for vdTan, vmdTan const MKL_Complex8* for vcTan, vmcTan const MKL_Complex16* for vzTan, vmzTan	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	float* for vsTan, vmsTan double* for vdTan, vmdTan MKL_Complex8* for vcTan, vmcTan MKL_Complex16* for vzTan, vmzTan	Pointer to an array that contains the output vector y .

Description

The $v?Tan$ function computes tangent of vector elements.

Note that arguments $abs(a[i]) \leq 2^{13}$ and $abs(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function $v?Tan(x)$

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\tan(z) = -i \cdot \tanh(i \cdot z).$$

Product and Performance Information

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

Notice revision #20201201

$v?Acos$

Computes inverse cosine of vector elements.

Syntax

```
vsAcos( n, a, y );
vsAcosI( n, a, inca, y, incy );
vmsAcos( n, a, y, mode );
vmsAcosI( n, a, inca, y, incy, mode );
vdAcos( n, a, y );
vdAcosI( n, a, inca, y, incy );
vmdAcos( n, a, y, mode );
vmdAcosI( n, a, inca, y, incy, mode );
```

```

vcAcos( n, a, y );
vcAcosI(n, a, inca, y, incy);
vmcAcos( n, a, y, mode );
vmcAcosI(n, a, inca, y, incy, mode);
vzAcos( n, a, y );
vzAcosI(n, a, inca, y, incy);
vmzAcos( n, a, y, mode );
vmzAcosI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const int	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsAcos, vmsAcos const double* for vdAcos, vmdAcos const MKL_Complex8* for vcAcos, vmcAcos const MKL_Complex16* for vzAcos, vmzAcos	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsAcos, vmsAcos double* for vdAcos, vmdAcos MKL_Complex8* for vcAcos, vmcAcos MKL_Complex16* for vzAcos, vmzAcos	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Acos` function computes inverse cosine of vector elements.

Special Values for Real Function `v?Acos(x)`

Argument	Result	VM Error Status	Exception
+0	$+\pi/2$		

Argument	Result	VM Error Status	Exception
-0	$+\pi/2$		
+1	+0		
-1	$+\pi$		
$ X > 1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function $v?Acos(z)$

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	$+3\pi/4-i\cdot\infty$	$+\pi/2-i\cdot\infty$	$+\pi/2-i\cdot\infty$	$+\pi/2-i\cdot\infty$	$+\pi/2-i\cdot\infty$	$+\pi/4-i\cdot\infty$	QNAN- $i\cdot\infty$
$+i\cdot Y$	$+\pi-i\cdot\infty$					$+0-i\cdot\infty$	QNAN $+i\cdot$ QNAN
$+i\cdot 0$	$+\pi-i\cdot\infty$		$+\pi/2-i\cdot 0$	$+\pi/2-i\cdot 0$		$+0-i\cdot\infty$	QNAN $+i\cdot$ QNAN
$-i\cdot 0$	$+\pi+i\cdot\infty$		$+\pi/2+i\cdot 0$	$+\pi/2+i\cdot 0$		$+0+i\cdot\infty$	QNAN $+i\cdot$ QNAN
$-i\cdot Y$	$+\pi+i\cdot\infty$					$+0+i\cdot\infty$	QNAN $+i\cdot$ QNAN
$-i\cdot\infty$	$+3\pi/4+i\cdot\infty$	$+\pi/2+i\cdot\infty$	$+\pi/2+i\cdot\infty$	$+\pi/2+i\cdot\infty$	$+\pi/2+i\cdot\infty$	$+\pi/4+i\cdot\infty$	QNAN+ $i\cdot\infty$
$+i\cdot$ NAN	QNAN+ $i\cdot\infty$	QNAN $+i\cdot$ QNAN	$+\pi/2+i\cdot$ QNAN	$+\pi/2+i\cdot$ QNAN	QNAN $+i\cdot$ QNAN	QNAN+ $i\cdot\infty$	QNAN $+i\cdot$ QNAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- $Acos(CONJ(z)) = CONJ(Acos(z))$.

$v?Asin$

Computes inverse sine of vector elements.

Syntax

```
vsAsin( n, a, y );
vsAsinI(n, a, inca, y, incy);
vmsAsin( n, a, y, mode );
vmsAsinI(n, a, inca, y, incy, mode);
vdAsin( n, a, y );
vdAsinI(n, a, inca, y, incy);
vmdAsin( n, a, y, mode );
vmdAsinI(n, a, inca, y, incy, mode);
vcAsin( n, a, y );
```

```
vcAsinI(n, a, inca, y, incy);
vmcAsin( n, a, y, mode );
vmcAsinI(n, a, inca, y, incy, mode);
vzAsin( n, a, y );
vzAsinI(n, a, inca, y, incy);
vmzAsin( n, a, y, mode );
vmzAsinI(n, a, inca, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsAsin, vmsAsin const double* for vdAsin, vmdAsin const MKL_Complex8* for vcAsin, vmcAsin const MKL_Complex16* for vzAsin, vmzAsin	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsAsin, vmsAsin double* for vdAsin, vmdAsin MKL_Complex8* for vcAsin, vmcAsin MKL_Complex16* for vzAsin, vmzAsin	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Asin` function computes inverse sine of vector elements.

Special Values for Real Function `v?Asin(x)`

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		

Argument	Result	VM Error Status	Exception
+1	$+\pi/2$		
-1	$-\pi/2$		
$ X > 1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$\text{Asin}(z) = -i \cdot \text{Asinh}(i \cdot z).$

v?Atan

Computes inverse tangent of vector elements.

Syntax

```

vsAtan( n, a, y );
vsAtanI(n, a, inca, y, incy);
vmsAtan( n, a, y, mode );
vmsAtanI(n, a, inca, y, incy, mode);
vdAtan( n, a, y );
vdAtanI(n, a, inca, y, incy);
vmdAtan( n, a, y, mode );
vmdAtanI(n, a, inca, y, incy, mode);
vcAtan( n, a, y );
vcAtanI(n, a, inca, y, incy);
vmcAtan( n, a, y, mode );
vmcAtanI(n, a, inca, y, incy, mode);
vzAtan( n, a, y );
vzAtanI(n, a, inca, y, incy);
vmzAtan( n, a, y, mode );
vmzAtanI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsAtan, vmsAtan	Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	const double* for vdAtan, vmdAtan	
	const MKL_Complex8* for vcAtan, vmcAtan	
	const MKL_Complex16* for vzAtan, vmzAtan	
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsAtan, vmsAtan double* for vdAtan, vmdAtan MKL_Complex8* for vcAtan, vmcAtan MKL_Complex16* for vzAtan, vmzAtan	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Atan` function computes inverse tangent of vector elements.

Special Values for Real Function `v?Atan(x)`

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
$+\infty$	$+\pi/2$		
$-\infty$	$-\pi/2$		
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Atan}(z) = -i * \text{Atanh}(i * z).$$

`v?Atan2`

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

```
vsAtan2( n, a, b, y );
vsAtan2I(n, a, inca, b, incb, y, incy);
vmsAtan2( n, a, b, y, mode );
vmsAtan2I(n, a, inca, b, incb, y, incy, mode);
vdAtan2( n, a, b, y );
```

```
vdAtan2I(n, a, inca, b, incb, y, incy);
vmdAtan2( n, a, b, y, mode );
vmdAtan2I(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a, b</i>	<code>const float*</code> for <code>vsAtan2</code> , <code>vmsAtan2</code> <code>const double*</code> for <code>vdAtan2</code> , <code>vmdAtan2</code>	Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> , <i>b</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsAtan2</code> , <code>vmsAtan2</code> <code>double*</code> for <code>vdAtan2</code> , <code>vmdAtan2</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Atan2` function computes four-quadrant inverse tangent of elements of two vectors.

The elements of the output vector *y* are computed as the four-quadrant arctangent of $a[i] / b[i]$.

Special values for Real Function `v?Atan2(x)`

Argument 1	Argument 2	Result	Exception
$-\infty$	$-\infty$	$-3*\pi/4$	
$-\infty$	$X < +0$	$-\pi/2$	
$-\infty$	-0	$-\pi/2$	
$-\infty$	$+0$	$-\pi/2$	
$-\infty$	$X > +0$	$-\pi/2$	
$-\infty$	$+\infty$	$-\pi/4$	
$X < +0$	$-\infty$	$-\pi$	
$X < +0$	-0	$-\pi/2$	
$X < +0$	$+0$	$-\pi/2$	
$X < +0$	$+\infty$	-0	
-0	$-\infty$	$-\pi$	

Argument 1	Argument 2	Result	Exception
-0	$X < +0$	$-\pi$	
-0	-0	$-\pi$	
-0	+0	-0	
-0	$X > +0$	-0	
-0	$+\infty$	-0	
+0	$-\infty$	$+\pi$	
+0	$X < +0$	$+\pi$	
+0	-0	$+\pi$	
+0	+0	+0	
+0	$X > +0$	+0	
+0	$+\infty$	+0	
$X > +0$	$-\infty$	$+\pi$	
$X > +0$	-0	$+\pi/2$	
$X > +0$	+0	$+\pi/2$	
$X > +0$	$+\infty$	+0	
$+\infty$	$-\infty$	$+3*\pi/4$	
$+\infty$	$X < +0$	$+\pi/2$	
$+\infty$	-0	$+\pi/2$	
$+\infty$	+0	$+\pi/2$	
$+\infty$	$X > +0$	$+\pi/2$	
$+\infty$	$+\infty$	$+\pi/4$	
$X > +0$	QNAN	QNAN	
$X > +0$	SNAN	QNAN	INVALID
QNAN	$X > +0$	QNAN	
SNAN	$X > +0$	QNAN	INVALID
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	INVALID
SNAN	QNAN	QNAN	INVALID
SNAN	SNAN	QNAN	INVALID

v?Cospi

Computes the cosine of vector elements multiplied by π .

Syntax

```

vsCospi (n, a, y);
vsCospiI(n, a, inca, y, incy);
vmsCospi (n, a, y, mode);
vmsCospiI(n, a, inca, y, incy, mode);
vdCospi (n, a, y);
vdCospiI(n, a, inca, y, incy);
vmdCospi (n, a, y, mode);
vmdCospiI(n, a, inca, y, incy, mode);

```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsCospi</code> <code>const float*</code> for <code>vmsCospi</code> <code>const double*</code> for <code>vdCospi</code> <code>const double*</code> for <code>vmdCospi</code>	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsCospi</code> <code>float*</code> for <code>vmsCospi</code> <code>double*</code> for <code>vdCospi</code> <code>double*</code> for <code>vmdCospi</code>	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Cospi` function computes the cosine of vector elements multiplied by *n*. For an argument *x*, the function computes $\cos(n*x)$.

Special values for Real Function `v?Cospi(x)`

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
<i>n</i> + 0.5, for any integer <i>n</i> where <i>n</i> + 0.5 is representable	+0		
$\pm\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Application Notes

If arguments $\text{abs}(a_i) \leq 2^{22}$ for single precision or $\text{abs}(a_i) \leq 2^{43}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

[Cos](#) Computes cosine of vector elements.

[Cosd](#) Computes the cosine of vector elements multiplied by $n/180$.

v?Sinpi

Computes the sine of vector elements multiplied by n .

Syntax

```
vsSinpi (n, a, y);
vsSinpiI(n, a, inca, y, incy);
vmsSinpi (n, a, y, mode);
vmsSinpiI(n, a, inca, y, incy, mode);
vdSinpi (n, a, y);
vdSinpiI(n, a, inca, y, incy);
vmdSinpi (n, a, y, mode);
vmdSinpiI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsSinpi</code> <code>const float*</code> for <code>vmsSinpi</code> <code>const double*</code> for <code>vdSinpi</code> <code>const double*</code> for <code>vmdSinpi</code>	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsSinpi	Pointer to an array containing the output vector <i>y</i> .
	float* for vmsSinpi	
	double* for vdSinpi	
	double* for vmdSinpi	

Description

The `v?Sinpi` function computes the sine of vector elements multiplied by n . For an argument x , the function computes $\sin(n \cdot x)$.

Special values for Real Function `v?Sinpi(x)`

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+ n , positive integer	+0		
- n , negative integer	-0		
$\pm\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Application Notes

If arguments $\text{abs}(a_i) \leq 2^{22}$ for single precision or $\text{abs}(a_i) \leq 2^{51}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

[Sin](#) Computes sine of vector elements.

[Sind](#) Computes the sine of vector elements multiplied by $n/180$.

`v?Tanpi`

Computes the tangent of vector elements multiplied by n .

Syntax

```
vsTanpi (n, a, y);
vsTanpiI(n, a, inca, y, incy);
vmsTanpi (n, a, y, mode);
vmsTanpiI(n, a, inca, y, incy, mode);
vdTanpi (n, a, y);
vdTanpiI(n, a, inca, y, incy);
vmdTanpi (n, a, y, mode);
vmdTanpiI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsTanpi</code> <code>const float*</code> for <code>vmsTanpi</code> <code>const double*</code> for <code>vdTanpi</code> <code>const double*</code> for <code>vmdTanpi</code>	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsTanpi</code> <code>float*</code> for <code>vmsTanpi</code> <code>double*</code> for <code>vdTanpi</code> <code>double*</code> for <code>vmdTanpi</code>	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Tanpi` function computes the tangent of vector elements multiplied by *n*. For an argument *x*, the function computes $\tan(n \cdot x)$.

Special values for Real Function `v?Tanpi(x)`

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
$\pm\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
<i>n</i> , even integer	<code>copysign(0.0, n)</code>		
<i>n</i> , odd integer	<code>copysign(0.0, -n)</code>		
<i>n</i> + 0.5, for <i>n</i> even integer and <i>n</i> + 0.5 representable	$+\infty$		
<i>n</i> + 0.5, for <i>n</i> odd integer and <i>n</i> + 0.5 representable	$-\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

The `copysign(x, y)` function returns the first vector argument *x* with the sign changed to match that of the second argument *y*.

Application Notes

If arguments $\text{abs}(a_i) \leq 2^{13}$ for single precision or $\text{abs}(a_i) \leq 2^{67}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

[Tan](#) Computes tangent of vector elements.

[Tand](#) Computes the tangent of vector elements multiplied by $\pi/180$.

v?Acospi

Computes the inverse cosine of vector elements divided by π .

Syntax

```
vsAcospi (n, a, y);
vsAcospiI(n, a, inca, y, incy);
vmsAcospi (n, a, y, mode);
vmsAcospiI(n, a, inca, y, incy, mode);
vdAcospi (n, a, y);
vdAcospiI(n, a, inca, y, incy);
vmdAcospi (n, a, y, mode);
vmdAcospiI(n, a, inca, y, incy, mode);
```

Include Files

- `mkh.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsAcospi</code> <code>const float*</code> for <code>vmsAcospi</code> <code>const double*</code> for <code>vdAcospi</code> <code>const double*</code> for <code>vmdAcospi</code>	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	<code>float*</code> for <code>vsAcosp</code>	Pointer to an array containing the output vector y .
	<code>float*</code> for <code>vmsAcosp</code>	
	<code>double*</code> for <code>vdAcosp</code>	
	<code>double*</code> for <code>vmdAcosp</code>	

Description

The `v?Acosp` function computes the inverse cosine of vector elements divided by n . For an argument x , the function computes $\text{acos}(x)/n$.

See [Special Value Notations](#) for the conventions used in this table:

Special values for Real Function `v?Acosp(x)`

Argument	Result	VM Error Status	Exception
+0	+1/2		
-0	+1/2		
+1	+0		
-1	+1		
$ x > 1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[Acos](#) Computes inverse cosine of vector elements.

`v?Asinpi`

Computes the inverse sine of vector elements divided by n .

Syntax

```
vsAsinpi (n, a, y);
vsAsinpiI(n, a, inca, y, incy);
vmsAsinpi (n, a, y, mode);
vmsAsinpiI(n, a, inca, y, incy, mode);
vdAsinpi (n, a, y);
vdAsinpiI(n, a, inca, y, incy);
vmdAsinpi (n, a, y, mode);
vmdAsinpiI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsAsinpi const float* for vmsAsinpi const double* for vdAsinpi const double* for vmdAsinpi	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsAsinpi float* for vmsAsinpi double* for vdAsinpi double* for vmdAsinpi	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Asinpi` function computes the inverse sine of vector elements divided by π . For an argument x , the function computes $\text{asin}(x)/\pi$.

Special values for Real Function `v?Asinpi(x)`

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+1	+1/2		
-1	-1/2		
$ x > 1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[Asin](#) Computes inverse sine of vector elements.

`v?Atanpi`

Computes the inverse tangent of vector elements divided by π .

Syntax

```
vsAtanpi (n, a, y);
vsAtanpiI(n, a, inca, y, incy);
vmsAtanpi (n, a, y, mode);
vmsAtanpiI(n, a, inca, y, incy, mode);
vdAtanpi (n, a, y);
vdAtanpiI(n, a, inca, y, incy);
vmdAtanpi (n, a, y, mode);
vmdAtanpiI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsAtanpi</code> <code>const float*</code> for <code>vmsAtanpi</code> <code>const double*</code> for <code>vdAtanpi</code> <code>const double*</code> for <code>vmdAtanpi</code>	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsAtanpi</code> <code>float*</code> for <code>vmsAtanpi</code> <code>double*</code> for <code>vdAtanpi</code> <code>double*</code> for <code>vmdAtanpi</code>	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Atanpi` function computes the inverse tangent of vector elements divided by π . For an argument x , the function computes $\text{atan}(x)/\pi$.

Special values for Real Function `v?Atanpi(x)`

Argument	Result	VM Error Status	Exception
+0	+0		

Argument	Result	VM Error Status	Exception
-0	-0		
$+\infty$	$+1/2$		
$-\infty$	$-1/2$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[Atan](#) Computes inverse tangent of vector elements.

v?Atan2pi

Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by n .

Syntax

```
vsAtan2pi (n, a, b, y);
vsAtan2piI(n, a, inca, b, incb, y, incy);
vmsAtan2pi (n, a, b, y, mode);
vmsAtan2piI(n, a, inca, b, incb, y, incy, mode);
vdAtan2pi (n, a, b, y);
vdAtan2piI(n, a, inca, b, incb, y, incy);
vmdAtan2pi (n, a, b, y, mode);
vmdAtan2piI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a, b</i>	<code>const float*</code> for <code>vsAtan2pi</code> <code>const float*</code> for <code>vmsAtan2pi</code> <code>const double*</code> for <code>vdAtan2pi</code> <code>const double*</code> for <code>vmdAtan2pi</code>	Pointers to the arrays containing the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> , <i>b</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	float* for vsAtan2pi float* for vmsAtan2pi double* for vdAtan2pi double* for vmdAtan2pi	Pointer to an array containing the output vector y .

Description

The `v?Atan2pi` function computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by n .

For the elements of the output vector y , the function computes the four-quadrant arctangent of a_i/b_i , with the result divided by n .

Special values for Real Function `v?Atan2pi(x, y)`

Argument 1	Argument 2	Result	Exception
$-\infty$	$-\infty$	$-3/4$	
$-\infty$	$x < +0$	$-1/2$	
$-\infty$	-0	$+1/2$	
$-\infty$	$+0$	$-1/2$	
$-\infty$	$x > +0$	$-1/2$	
$-\infty$	$+\infty$	$-1/4$	
$x < +0$	$-\infty$	-1	
$x < +0$	-0	$-1/2$	
$x < +0$	$+0$	$-1/2$	
$x < +0$	$+\infty$	-0	
-0	$-\infty$	-1	
-0	$x < +0$	-1	
-0	-0	-1	
-0	$+0$	-0	
-0	$x > +0$	-0	
-0	$+\infty$	-0	
$+0$	$-\infty$	$+1$	
$+0$	$x < +0$	$+1$	
$+0$	-0	$+1$	
$+0$	$+0$	$+0$	
$+0$	$x > +0$	$+0$	
$+0$	$+\infty$	$+0$	
$x > +0$	$-\infty$	$+1$	
$x > +0$	-0	$+1/2$	
$x > +0$	$+0$	$+1/2$	
$x > +0$	$+\infty$	$+1/4$	
$+\infty$	$-\infty$	$+3/4$	
$+\infty$	$x < +0$	$+1/2$	
$+\infty$	-0	$+1/2$	
$+\infty$	$+0$	$+1/2$	

Argument 1	Argument 2	Result	Exception
$+\infty$	$x > +0$	$+1/2$	
$+\infty$	$+\infty$	$+1/4$	
$x > +0$	QNAN	QNAN	
$x > +0$	SNAN	QNAN	INVALID
QNAN	$x > +0$	QNAN	
SNAN	$x > +0$	QNAN	INVALID
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	INVALID
SNAN	QNAN	QNAN	INVALID
SNAN	SNAN	QNAN	INVALID

See Also

[Atan2](#) Computes four-quadrant inverse tangent of elements of two vectors.

v?Cosd

Computes the cosine of vector elements multiplied by $n/180$.

Syntax

```
vsCosd (n, a, y);
vsCosdI(n, a, inca, y, incy);
vmsCosd (n, a, y, mode);
vmsCosdI(n, a, inca, y, incy, mode);
vdCosd (n, a, y);
vdCosdI(n, a, inca, y, incy);
vmdCosd (n, a, y, mode);
vmdCosdI(n, a, inca, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsCosd const float* for vmsCosd const double* for vdCosd const double* for vmdCosd	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .

Name	Type	Description
<i>mode</i>	const MKL_INT64	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsCosd float* for vmsCosd double* for vdCosd double* for vmdCosd	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Cosd` function computes the cosine of vector elements multiplied by $n/180$. For an argument x , the function computes $\cos(n*x/180)$.

Special values for Real Function v?Cosd(x)

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
$\pm\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Application Notes

If arguments $\text{abs}(a_i) \leq 2^{24}$ for single precision or $\text{abs}(a_i) \leq 2^{52}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

[Cos](#) Computes cosine of vector elements.

[Cospi](#) Computes the cosine of vector elements multiplied by n .

v?Sind

Computes the sine of vector elements multiplied by $n/180$.

Syntax

```
vsSind (n, a, y);
```

```
vsSindI(n, a, inca, y, incy);
```

```
vmsSind (n, a, y, mode);
```

```
vmsSindI(n, a, inca, y, incy, mode);
```

```
vdSind (n, a, y);
```

```
vdSindI(n, a, inca, y, incy);
```

```
vmdSind (n, a, y, mode);
vmdSindI(n, a, inca, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsSind const float* for vmsSind const double* for vdSind const double* for vmdSind	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsSind float* for vmsSind double* for vdSind double* for vmdSind	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Sind` function computes the sine of vector elements multiplied by $\pi/180$. For an argument *x*, the function computes $\sin(\pi*x/180)$.

Special values for Real Function v?Sind(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
$\pm\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Application Notes

If arguments $\text{abs}(a_i) \leq 2^{24}$ for single precision or $\text{abs}(a_i) \leq 2^{52}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

[Sin](#) Computes sine of vector elements.

[Sinpi](#) Computes the sine of vector elements multiplied by π .

v?Tand

Computes the tangent of vector elements multiplied by $\pi/180$.

Syntax

```
vsTand (n, a, y);
vsTandI(n, a, inca, y, incy);
vmsTand (n, a, y, mode);
vmsTandI(n, a, inca, y, incy, mode);
vdTand (n, a, y);
vdTandI(n, a, inca, y, incy);
vmdTand (n, a, y, mode);
vmdTandI(n, a, inca, y, incy, mode);
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsTand</code> <code>const float*</code> for <code>vmsTand</code> <code>const double*</code> for <code>vdTand</code> <code>const double*</code> for <code>vmdTand</code>	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	<code>float*</code> for <code>vsTand</code>	Pointer to an array containing the output vector y .
	<code>float*</code> for <code>vmsTand</code>	
	<code>double*</code> for <code>vdTand</code>	
	<code>double*</code> for <code>vmdTand</code>	

Description

The `v?Tand` function computes the tangent of vector elements multiplied by $\pi/180$. For an argument x , the function computes $\tan(\pi*x/180)$.

Special values for Real Function `v?Tand(x)`

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
$\pm\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

The `copysign(x, y)` function returns the first vector argument x with the sign changed to match that of the second argument y .

Application Notes

If arguments $\text{abs}(a_i) \leq 2^{38}$ for single precision or $\text{abs}(a_i) \leq 2^{67}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

See Also

[Tan](#) Computes tangent of vector elements.

[Tanpi](#) Computes the tangent of vector elements multiplied by π .

Hyperbolic Functions

`v?Cosh`

Computes hyperbolic cosine of vector elements.

Syntax

```
vsCosh( n, a, y );
vsCoshI(n, a, inca, y, incy);
vmsCosh( n, a, y, mode );
vmsCoshI(n, a, inca, y, incy, mode);
vdCosh( n, a, y );
vdCoshI(n, a, inca, y, incy);
vmdCosh( n, a, y, mode );
```



```

vmdCoshI(n, a, inca, y, incy, mode);
vcCosh( n, a, y );
vcCoshI(n, a, inca, y, incy);
vmcCosh( n, a, y, mode );
vmcCoshI(n, a, inca, y, incy, mode);
vzCosh( n, a, y );
vzCoshI(n, a, inca, y, incy);
vmzCosh( n, a, y, mode );
vmzCoshI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsCosh, vmsCosh const double* for vdCosh, vmdCosh const MKL_Complex8* for vcCosh, vmcCosh const MKL_Complex16* for vzCosh, vmzCosh	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Cosh Function

Data Type	Threshold Limitations on Input Parameters
single precision	$-\ln(\text{FLT_MAX}) - \ln 2 < a[i] < \ln(\text{FLT_MAX}) + \ln 2$
double precision	$-\ln(\text{DBL_MAX}) - \ln 2 < a[i] < \ln(\text{DBL_MAX}) + \ln 2$

Precision overflow thresholds for the complex *v?Cosh* function are beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsCosh, vmsCosh double* for vdCosh, vmdCosh MKL_Complex8* for vcCosh, vmcCosh	Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
------	------	-------------

MKL_Complex16* for vzCosh,
vmzCosh

Description

The `v?Cosh` function computes hyperbolic cosine of vector elements.

Special Values for Real Function `v?Cosh(x)`

Argument	Result	VM Error Status	Exception
+0	+1		
-0	+1		
X > overflow	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
X < -overflow	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
$+\infty$	$+\infty$		
$-\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function `v?Cosh(z)`

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	$+\infty$ $+i\cdot\text{QNAN}$ INVALID	QNAN $+i\cdot\text{QNAN}$ INVALID	QNAN- $i\cdot 0$ INVALID	QNAN+ $i\cdot 0$ INVALID	QNAN $+i\cdot\text{QNAN}$ INVALID	$+\infty$ $+i\cdot\text{QNAN}$ INVALID	QNAN $+i\cdot\text{QNAN}$
$+i\cdot Y$	$+\infty\cdot\text{Cos}(Y)$ - $i\cdot\infty\cdot\text{Sin}(Y)$					$+\infty\cdot\text{CIS}(Y)$	QNAN $+i\cdot\text{QNAN}$
$+i\cdot 0$	$+\infty-i\cdot 0$		$+1-i\cdot 0$	$+1+i\cdot 0$		$+\infty+i\cdot 0$	QNAN+ $i\cdot 0$
$-i\cdot 0$	$+\infty+i\cdot 0$		$+1+i\cdot 0$	$+1-i\cdot 0$		$+\infty-i\cdot 0$	QNAN- $i\cdot 0$
$-i\cdot Y$	$+\infty\cdot\text{Cos}(Y)$ - $i\cdot\infty\cdot\text{Sin}(Y)$					$+\infty\cdot\text{CIS}(Y)$	QNAN $+i\cdot\text{QNAN}$
$-i\cdot\infty$	$+\infty$ $+i\cdot\text{QNAN}$ INVALID	QNAN $+i\cdot\text{QNAN}$ INVALID	QNAN+ $i\cdot 0$ INVALID	QNAN- $i\cdot 0$ INVALID	QNAN $+i\cdot\text{QNAN}$ INVALID	$+\infty$ $+i\cdot\text{QNAN}$ INVALID	QNAN $+i\cdot\text{QNAN}$
$+i\cdot\text{NAN}$	$+\infty$ $+i\cdot\text{QNAN}$	QNAN $+i\cdot\text{QNAN}$	QNAN $+i\cdot\text{QNAN}$	QNAN- $i\cdot\text{QNAN}$	QNAN $+i\cdot\text{QNAN}$	$+\infty$ $+i\cdot\text{QNAN}$	QNAN $+i\cdot\text{QNAN}$

Notes:

- raises the `INVALID` exception when the real or imaginary part of the argument is `SNAN`
- raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW` in the case of overflow, that is, when `RE(z)`, `IM(z)` are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- $\text{Cosh}(\text{CONJ}(z)) = \text{CONJ}(\text{Cosh}(z))$
- $\text{Cosh}(-z) = \text{Cosh}(z)$.

v?Sinh*Computes hyperbolic sine of vector elements.***Syntax**

```

vsSinh( n, a, y );
vsSinhI(n, a, inca, y, incy);
vmsSinh( n, a, y, mode );
vmsSinhI(n, a, inca, y, incy, mode);
vdSinh( n, a, y );
vdSinhI(n, a, inca, y, incy);
vmdSinh( n, a, y, mode );
vmdSinhI(n, a, inca, y, incy, mode);
vcSinh( n, a, y );
vcSinhI(n, a, inca, y, incy);
vmcSinh( n, a, y, mode );
vmcSinhI(n, a, inca, y, incy, mode);
vzSinh( n, a, y );
vzSinhI(n, a, inca, y, incy);
vmzSinh( n, a, y, mode );
vmzSinhI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsSinh, vmsSinh const double* for vdSinh, vmdSinh const MKL_Complex8* for vcSinh, vmcSinh const MKL_Complex16* for vzSinh, vmzSinh	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Sinh Function

Data Type	Threshold Limitations on Input Parameters
single precision	$-\ln(\text{FLT_MAX}) - \ln 2 < a[i] < \ln(\text{FLT_MAX}) + \ln 2$
double precision	$-\ln(\text{DBL_MAX}) - \ln 2 < a[i] < \ln(\text{DBL_MAX}) + \ln 2$

Precision overflow thresholds for the complex v?Sinh function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	float* for vsSinh, vmsSinh double* for vdSinh, vmdSinh MKL_Complex8* for vcSinh, vmcSinh MKL_Complex16* for vzSinh, vmzSinh	Pointer to an array that contains the output vector y .

Description

The v?Sinh function computes hyperbolic sine of vector elements.

Special Values for Real Function v?Sinh(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
$X > \text{overflow}$	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
$X < -\text{overflow}$	$-\infty$	VML_STATUS_OVERFLOW	OVERFLOW
$+\infty$	$+\infty$		
$-\infty$	$-\infty$		
QNaN	QNaN		
SNAN	QNaN		INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function v?Sinh(z)

$\text{RE}(z)$ $i \cdot \text{IM}(z)$	$-\infty$	$-X$	-0	$+0$	$+X$	$+\infty$	NAN
$+i \cdot \infty$	$-\infty + i \cdot \text{QNaN}$ INVALID	QNaN $+i \cdot \text{QNaN}$ INVALID	$-0 + i \cdot \text{QNaN}$ INVALID	$+0 + i \cdot \text{QNaN}$ INVALID	QNaN $+i \cdot \text{QNaN}$ INVALID	$+\infty + i \cdot \text{QNaN}$ INVALID	QNaN $+i \cdot \text{QNaN}$
$+i \cdot Y$	$-\infty \cdot \cos(Y) + i \cdot \infty \cdot \sin(Y)$					$+\infty \cdot \text{CIS}(Y)$	QNaN $+i \cdot \text{QNaN}$
$+i \cdot 0$	$-\infty + i \cdot 0$		$-0 + i \cdot 0$	$+0 + i \cdot 0$		$+\infty + i \cdot 0$	QNaN $+i \cdot 0$
$-i \cdot 0$	$-\infty - i \cdot 0$		$-0 - i \cdot 0$	$+0 - i \cdot 0$		$+\infty - i \cdot 0$	QNaN $-i \cdot 0$
$-i \cdot Y$	$-\infty \cdot \cos(Y) + i \cdot \infty \cdot \sin(Y)$					$+\infty \cdot \text{CIS}(Y)$	QNaN $+i \cdot \text{QNaN}$

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
-i·∞	-∞+i·QNAN INVALID	QNAN +i·QNAN INVALID	-0+i·QNAN INVALID	+0+i·QNAN N INVALID	QNAN +i·QNAN INVALID	+∞ +i·QNAN INVALID	QNAN +i·QNAN
+i·NAN	-∞+i·QNAN	QNAN +i·QNAN	-0+i·QNAN	+0+i·QNAN N	QNAN +i·QNAN	+∞ +i·QNAN	QNAN +i·QNAN

Notes:

- raises the `INVALID` exception when the real or imaginary part of the argument is `SNAN`
- raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW` in the case of overflow, that is, when `RE(z)`, `IM(z)` are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- `Sinh(CONJ(z))=CONJ(Sinh(z))`
- `Sinh(-z)=-Sinh(z)`.

v?Tanh

Computes hyperbolic tangent of vector elements.

Syntax

```
vsTanh( n, a, y );
vsTanhI(n, a, inca, y, incy);
vmsTanh( n, a, y, mode );
vmsTanhI(n, a, inca, y, incy, mode);
vdTanh( n, a, y );
vdTanhI(n, a, inca, y, incy);
vmdTanh( n, a, y, mode );
vmdTanhI(n, a, inca, y, incy, mode);
vcTanh( n, a, y );
vcTanhI(n, a, inca, y, incy);
vmcTanh( n, a, y, mode );
vmcTanhI(n, a, inca, y, incy, mode);
vzTanh( n, a, y );
vzTanhI(n, a, inca, y, incy);
vmzTanh( n, a, y, mode );
vmzTanhI(n, a, inca, y, incy, mode);
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsTanh, vmsTanh const double* for vdTanh, vmdTanh const MKL_Complex8* for vcTanh, vmcTanh const MKL_Complex16* for vzTanh, vmzTanh	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsTanh, vmsTanh double* for vdTanh, vmdTanh MKL_Complex8* for vcTanh, vmcTanh MKL_Complex16* for vzTanh, vmzTanh	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Tanh` function computes hyperbolic tangent of vector elements.

Special Values for Real Function `v?Tanh(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	+1	
$-\infty$	-1	
QNaN	QNaN	
SNAN	QNaN	INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function `v?Tanh(z)`

RE(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
i·IM(z)							
$+i\cdot\infty$	-1+i·0	QNaN +i·QNaN INVALID	QNaN +i·QNaN INVALID	QNaN +i·QNaN INVALID	QNaN +i·QNaN INVALID	+1+i·0	QNaN +i·QNaN

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·Y	-1+i·0·Tan(Y)					+1+i·0·Tan(Y)	QNAN +i·QNAN
+i·0	-1+i·0		-0+i·0	+0+i·0		+1+i·0	QNAN+i·0
-i·0	-1-i·0		-0-i·0	+0-i·0		+1-i·0	QNAN-i·0
-i·Y	-1+i·0·Tan(Y)					+1+i·0·Tan(Y)	QNAN +i·QNAN
-i·∞	-1-i·0	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	QNAN +i·QNAN INVALID	+1-i·0	QNAN +i·QNAN
+i·NAN	-1+i·0	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	+1+i·0	QNAN +i·QNAN

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- $\text{Tanh}(\text{CONJ}(z)) = \text{CONJ}(\text{Tanh}(z))$
- $\text{Tanh}(-z) = -\text{Tanh}(z)$.

v?Acosh

Computes inverse hyperbolic cosine (nonnegative) of vector elements.

Syntax

```
vsAcosh( n, a, y );
vsAcoshI(n, a, inca, y, incy);
vmsAcosh( n, a, y, mode );
vmsAcoshI(n, a, inca, y, incy, mode);
vdAcosh( n, a, y );
vdAcoshI(n, a, inca, y, incy);
vmdAcosh( n, a, y, mode );
vmdAcoshI(n, a, inca, y, incy, mode);
vcAcosh( n, a, y );
vcAcoshI(n, a, inca, y, incy);
vmcAcosh( n, a, y, mode );
vmcAcoshI(n, a, inca, y, incy, mode);
vzAcosh( n, a, y );
vzAcoshI(n, a, inca, y, incy);
vmzAcosh( n, a, y, mode );
vmzAcoshI(n, a, inca, y, incy, mode);
```

Include Files

- `mkL.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsAcosh</code> , <code>vmsAcosh</code> <code>const double*</code> for <code>vdAcosh</code> , <code>vmdAcosh</code> <code>const MKL_Complex8*</code> for <code>vcAcosh</code> , <code>vmcAcosh</code> <code>const MKL_Complex16*</code> for <code>vzAcosh</code> , <code>vmzAcosh</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>cnst MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsAcosh</code> , <code>vmsAcosh</code> <code>double*</code> for <code>vdAcosh</code> , <code>vmdAcosh</code> <code>MKL_Complex8*</code> for <code>vcAcosh</code> , <code>vmcAcosh</code> <code>MKL_Complex16*</code> for <code>vzAcosh</code> , <code>vmzAcosh</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Acosh` function computes inverse hyperbolic cosine (nonnegative) of vector elements.

Special Values for Real Function `v?Acosh(x)`

Argument	Result	VM Error Status	Exception
+1	+0		
$X < +1$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function $v?Acosh(z)$

RE(z) i·IM(z)	$-\infty$	-X	-0	+0	+X	$+\infty$	NAN
$+i\cdot\infty$	$+\infty + i \cdot \frac{3\pi}{4}$	$+\infty + i\cdot\pi/2$	$+\infty + i\cdot\pi/2$	$+\infty + i\cdot\pi/2$	$+\infty + i\cdot\pi/2$	$+\infty + i\cdot\pi/4$	$+\infty + i\cdot QNAN$
$+i\cdot Y$	$+\infty + i\cdot\pi$					$+\infty + i\cdot 0$	QNAN $+i\cdot QNAN$
$+i\cdot 0$	$+\infty + i\cdot\pi$		$+0 + i\cdot\pi/2$	$+0 + i\cdot\pi/2$		$+\infty + i\cdot 0$	QNAN $+i\cdot QNAN$
$-i\cdot 0$	$+\infty + i\cdot\pi$		$+0 + i\cdot\pi/2$	$+0 + i\cdot\pi/2$		$+\infty + i\cdot 0$	QNAN $+i\cdot QNAN$
$-i\cdot Y$	$+\infty + i\cdot\pi$					$+\infty + i\cdot 0$	QNAN $+i\cdot QNAN$
$-i\cdot\infty$	$+\infty - i \cdot \frac{3\pi}{4}$	$+\infty - i\cdot\pi/2$	$+\infty - i\cdot\pi/2$	$+\infty - i\cdot\pi/2$	$+\infty - i\cdot\pi/2$	$+\infty - i\cdot\pi/4$	$+\infty + i\cdot QNAN$
$+i\cdot NAN$	$+\infty$ $+i\cdot QNAN$	QNAN $+i\cdot QNAN$	QNAN $+i\cdot QNAN$	QNAN $+i\cdot QNAN$	QNAN $+i\cdot QNAN$	$+\infty$ $+i\cdot QNAN$	QNAN $+i\cdot QNAN$

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- $Acosh(CONJ(z)) = CONJ(Acosh(z))$.

 $v?Asinh$ Computes inverse hyperbolic sine of vector elements.**Syntax**

```

vsAsinh( n, a, y );
vsAsinhI(n, a, inca, y, incy);
vmsAsinh( n, a, y, mode );
vmsAsinhI(n, a, inca, y, incy, mode);
vdAsinh( n, a, y );
vdAsinhI(n, a, inca, y, incy);
vmdAsinh( n, a, y, mode );
vmdAsinhI(n, a, inca, y, incy, mode);
vcAsinh( n, a, y );
vcAsinhI(n, a, inca, y, incy);
vmcAsinh( n, a, y, mode );
vmcAsinhI(n, a, inca, y, incy, mode);
vzAsinh( n, a, y );

```

```

vzAsinhI(n, a, inca, y, incy);
vmzAsinh( n, a, y, mode );
vmzAsinhI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsAsinh, vmsAsinh const double* for vdAsinh, vmdAsinh const MKL_Complex8* for vcAsinh, vmcAsinh const MKL_Complex16* for vzAsinh, vmzAsinh	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsAsinh, vmsAsinh double* for vdAsinh, vmdAsinh MKL_Complex8* for vcAsinh, vmcAsinh MKL_Complex16* for vzAsinh, vmzAsinh	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Asinh` function computes inverse hyperbolic sine of vector elements.

Special Values for Real Function v?Asinh(x)

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
QNAN	QNAN	

Argument	Result	Exception
SNAN	QNAN	INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function $v?Asinh(z)$

$RE(z)$ $i \cdot IM(z)$	$-\infty$	$-X$	-0	$+0$	$+X$	$+\infty$	NAN
$+i \cdot \infty$	$-\infty + i \cdot \pi/4$	$-\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/4$	$+\infty + i \cdot QNAN$
$+i \cdot Y$	$-\infty + i \cdot 0$					$+\infty + i \cdot 0$	QNAN $+i \cdot QNAN$
$+i \cdot 0$	$+\infty + i \cdot 0$		$+0 + i \cdot 0$	$+0 + i \cdot 0$		$+\infty + i \cdot 0$	QNAN $+i \cdot QNAN$
$-i \cdot 0$	$-\infty - i \cdot 0$		$-0 - i \cdot 0$	$+0 - i \cdot 0$		$+\infty - i \cdot 0$	QNAN- $i \cdot QNAN$
$-i \cdot Y$	$-\infty - i \cdot 0$					$+\infty - i \cdot 0$	QNAN $+i \cdot QNAN$
$-i \cdot \infty$	$-\infty - i \cdot \pi/4$	$-\infty - i \cdot \pi/2$	$+\infty - i \cdot \pi/2$	$+\infty - i \cdot \pi/2$	$+\infty - i \cdot \pi/2$	$+\infty - i \cdot \pi/4$	$+\infty + i \cdot QNAN$
$+i \cdot NAN$	$-\infty + i \cdot QNAN$	QNAN $+i \cdot QNAN$	QNAN $+i \cdot QNAN$	QNAN $+i \cdot QNAN$	QNAN $+i \cdot QNAN$	$+\infty + i \cdot QNAN$	QNAN $+i \cdot QNAN$

Notes:

- raises `INVALID` exception when real or imaginary part of the argument is `SNAN`
- $Asinh(CONJ(z)) = CONJ(Asinh(z))$
- $Asinh(-z) = -Asinh(z)$.

$v?Atanh$

Computes inverse hyperbolic tangent of vector elements.

Syntax

```

vsAtanh( n, a, y );
vsAtanhI(n, a, inca, y, incy);
vmsAtanh( n, a, y, mode );
vmsAtanhI(n, a, inca, y, incy, mode);
vdAtanh( n, a, y );
vdAtanhI(n, a, inca, y, incy);
vmdAtanh( n, a, y, mode );
vmdAtanhI(n, a, inca, y, incy, mode);
vcAtanh( n, a, y );
vcAtanhI(n, a, inca, y, incy);
vmcAtanh( n, a, y, mode );
vmcAtanhI(n, a, inca, y, incy, mode);

```

```

vzAtanh( n, a, y );
vzAtanhI(n, a, inca, y, incy);
vmzAtanh( n, a, y, mode );
vmzAtanhI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsAtanh, vmsAtanh const double* for vdAtanh, vmdAtanh const MKL_Complex8* for vcAtanh, vmcAtanh const MKL_Complex16* for vzAtanh, vmzAtanh	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsAtanh, vmsAtanh double* for vdAtanh, vmdAtanh MKL_Complex8* for vcAtanh, vmcAtanh MKL_Complex16* for vzAtanh, vmzAtanh	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Atanh` function computes inverse hyperbolic tangent of vector elements.

Special Values for Real Function `v?Atanh(x)`

Argument	Result	VM Error Status	Exception
+1	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-1	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$ X > 1$	QNAN	VML_STATUS_ERRDOM	INVALID

Argument	Result	VM Error Status	Exception
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See [Special Value Notations](#) for the conventions used in the table below.

Special Values for Complex Function $v?Atanh(z)$

$RE(z)$ $i \cdot IM(z)$	$-\infty$	$-X$	-0	$+0$	$+X$	$+\infty$	NAN
$+i \cdot \infty$	$-0+i \cdot \pi/2$	$-0+i \cdot \pi/2$	$-0+i \cdot \pi/2$	$+0+i \cdot \pi/2$	$+0+i \cdot \pi/2$	$+0+i \cdot \pi/2$	$+0+i \cdot \pi/2$
$+i \cdot Y$	$-0+i \cdot \pi/2$					$+0+i \cdot \pi/2$	QNAN $+i \cdot QNAN$
$+i \cdot 0$	$-0+i \cdot \pi/2$		$-0+i \cdot 0$	$+0+i \cdot 0$		$+0+i \cdot \pi/2$	QNAN $+i \cdot QNAN$
$-i \cdot 0$	$-0-i \cdot \pi/2$		$-0-i \cdot 0$	$+0-i \cdot 0$		$+0-i \cdot \pi/2$	QNAN- $i \cdot QNAN$
$-i \cdot Y$	$-0-i \cdot \pi/2$					$+0-i \cdot \pi/2$	QNAN $+i \cdot QNAN$
$-i \cdot \infty$	$-0-i \cdot \pi/2$	$-0-i \cdot \pi/2$	$-0-i \cdot \pi/2$	$+0-i \cdot \pi/2$	$+0-i \cdot \pi/2$	$+0-i \cdot \pi/2$	$+0-i \cdot \pi/2$
$+i \cdot NAN$	$-0+i \cdot QNAN$	QNAN $+i \cdot QNAN$	$-0+i \cdot QNAN$	$+0+i \cdot QNAN$	QNAN $+i \cdot QNAN$	$+0+i \cdot QNAN$	QNAN $+i \cdot QNAN$

Notes:

- $Atanh(+/-1-i \cdot 0) = +/-\infty - i \cdot 0$, and ZERODIVIDE exception is raised
- raises INVALID exception when real or imaginary part of the argument is SNAN
- $Atanh(CONJ(z)) = CONJ(Atanh(z))$
- $Atanh(-z) = -Atanh(z)$.

Special Functions

$v?Erf$

Computes the error function value of vector elements.

Syntax

```
vsErf( n, a, y );
vsErfI( n, a, inca, y, incy );
vmsErf( n, a, y, mode );
vmsErfI( n, a, inca, y, incy, mode );
vdErf( n, a, y );
vdErfI( n, a, inca, y, incy );
vmdErf( n, a, y, mode );
```

```
vmdErfI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsErf</code> , <code>vmsErf</code> <code>const double*</code> for <code>vdErf</code> , <code>vmdErf</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca</i> , <i>incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsErf</code> , <code>vmsErf</code> <code>double*</code> for <code>vdErf</code> , <code>vmdErf</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `Erf` function computes the error function values for elements of the input vector *a* and writes them to the output vector *y*.

The error function is defined as given by:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

$$1. \quad \text{erfc}(x) = 1 - \text{erf}(x),$$

where `erfc` is the complementary error function.

$$2. \quad \Phi(x) = \frac{1}{2} \text{erf}(x/\sqrt{2}),$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

is the cumulative normal distribution function.

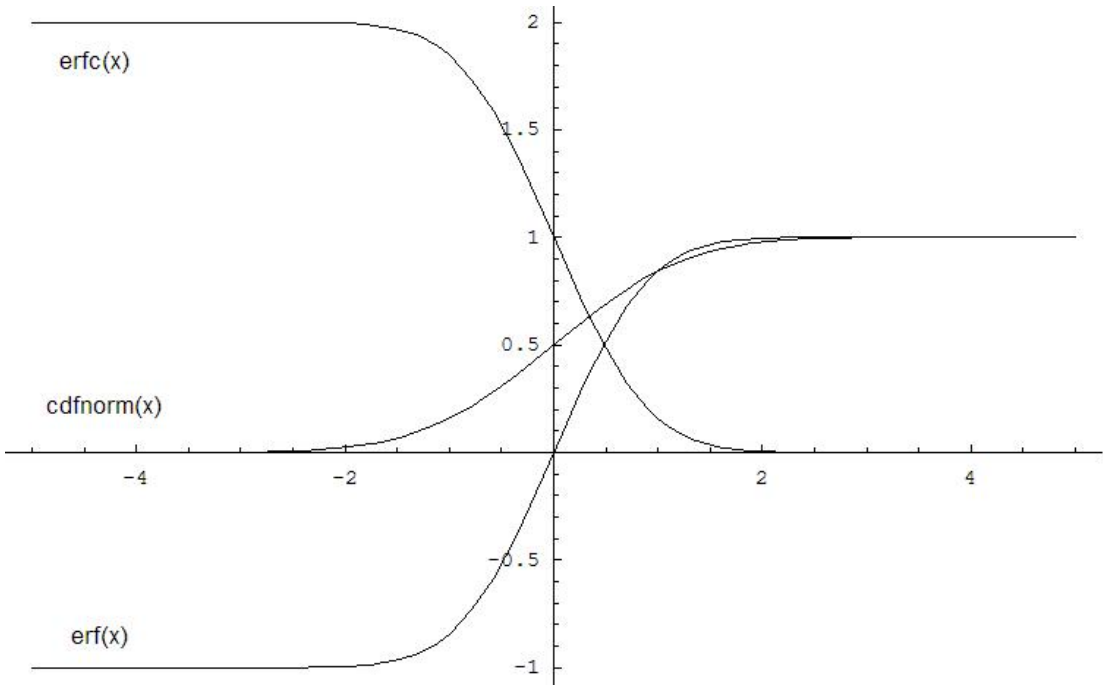
3. $\Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1)$,

where $\Phi^{-1}(x)$ and $\operatorname{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\operatorname{erf}(x)$ respectively.

The following figure illustrates the relationships among Erf family functions (Erf, Erfc, CdfNorm).

__border__top

Erf Family Functions Relationship



Useful relations for these functions:

$$\operatorname{erf}(x) + \operatorname{erfc}(x) = 1$$

$$\operatorname{cdfnorm}(x) = \frac{1}{2} \left(1 + \operatorname{erf} \left(\frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \operatorname{erfc} \left(\frac{x}{\sqrt{2}} \right)$$

Special Values for Real Function v?Erf(x)

Argument	Result	Exception
$+\infty$	+1	
$-\infty$	-1	
QNAN	QNAN	

Argument	Result	Exception
SNAN	QNAN	INVALID

See Also

[Erfc](#)

[CdfNorm](#)

v?Erfc

Computes the complementary error function value of vector elements.

Syntax

```
vsErfc( n, a, y );
vsErfcI(n, a, inca, y, incy);
vmsErfc( n, a, y, mode );
vmsErfcI(n, a, inca, y, incy, mode);
vdErfc( n, a, y );
vdErfcI(n, a, inca, y, incy);
vmdErfc( n, a, y, mode );
vmdErfcI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsErfc</code> , <code>vmsErfc</code> <code>const double*</code> for <code>vdErfc</code> , <code>vmdErfc</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsErfc</code> , <code>vmsErfc</code> <code>double*</code> for <code>vdErfc</code> , <code>vmdErfc</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `Erfc` function computes the complementary error function values for elements of the input vector *a* and writes them to the output vector *y*.

The complementary error function is defined as follows:

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt .$$

Useful relations:

1. $\text{erfc}(x) = 1 - \text{erf}(x)$.
2. $\Phi(x) = \frac{1}{2} \text{erf}(x/\sqrt{2})$,

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

is the cumulative normal distribution function.

3. $\Phi^{-1}(x) = \sqrt{2} \text{erf}^{-1}(2x - 1)$,

where $\Phi^{-1}(x)$ and $\text{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\text{erf}(x)$ respectively.

See also [Figure "Erf Family Functions Relationship"](#) in `Erf` function description for `Erfc` function relationship with the other functions of `Erf` family.

Special Values for Real Function `v?Erfc(x)`

Argument	Result	VM Error Status	Exception
$X > \text{underflow}$	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
$+\infty$	+0		
$-\infty$	+2		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[Erf](#)

[CdfNorm](#)

`v?CdfNorm`

Computes the cumulative normal distribution function values of vector elements.

Syntax

```
vsCdfNorm( n, a, y );
```

```

vsCdfNormI(n, a, inca, y, incy);
vmsCdfNorm( n, a, y, mode );
vmsCdfNormI(n, a, inca, y, incy, mode);
vdCdfNorm( n, a, y );
vdCdfNormI(n, a, inca, y, incy);
vmdCdfNorm( n, a, y, mode );
vmdCdfNormI(n, a, inca, y, incy, mode);

```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsCdfNorm</code> , <code>vmsCdfNorm</code> <code>const double*</code> for <code>vdCdfNorm</code> , <code>vmdCdfNorm</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsCdfNorm</code> , <code>vmsCdfNorm</code> <code>double*</code> for <code>vdCdfNorm</code> , <code>vmdCdfNorm</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `CdfNorm` function computes the cumulative normal distribution function values for elements of the input vector *a* and writes them to the output vector *y*.

The cumulative normal distribution function is defined as given by:

$$\text{CdfNorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt .$$

Useful relations:

$$\text{cdfnorm}(x) = \frac{1}{2} \left(1 + \text{erf} \left(\frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \text{erfc} \left(\frac{x}{\sqrt{2}} \right)$$

where `Erf` and `Erfc` are the error and complementary error functions.

See also [Figure "Erf Family Functions Relationship"](#) in `Erf` function description for `CdfNorm` function relationship with the other functions of `Erf` family.

Special Values for Real Function `v?CdfNorm(x)`

Argument	Result	VM Error Status	Exception
$X < \text{underflow}$	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
$+\infty$	+1		
$-\infty$	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[Erf](#)

[Erfc](#)

`v?ErfInv`

Computes inverse error function value of vector elements.

Syntax

```
vsErfInv( n, a, y );
vsErfInvI(n, a, inca, y, incy);
vmsErfInv( n, a, y, mode );
vmsErfInvI(n, a, inca, y, incy, mode);
vdErfInv( n, a, y );
vdErfInvI(n, a, inca, y, incy);
vmdErfInv( n, a, y, mode );
vmdErfInvI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsErfInv</code> , <code>vmsErfInv</code>	Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	<code>const double*</code> for <code>vdErfInv</code> , <code>vmdErfInv</code>	
<code>inca, incy</code>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<code>mode</code>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsErfInv</code> , <code>vmsErfInv</code> <code>double*</code> for <code>vdErfInv</code> , <code>vmdErfInv</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `ErfInv` function computes the inverse error function values for elements of the input vector *a* and writes them to the output vector *y*

$$y = \text{erf}^{-1}(a),$$

where `erf(x)` is the error function defined as given by:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

$$1. \text{erf}^{-1}(x) = \text{erfc}^{-1}(1 - x),$$

where `erfc` is the complementary error function.

$$2. \Phi(x) = \frac{1}{2} \text{erf}(x/\sqrt{2}),$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

is the cumulative normal distribution function.

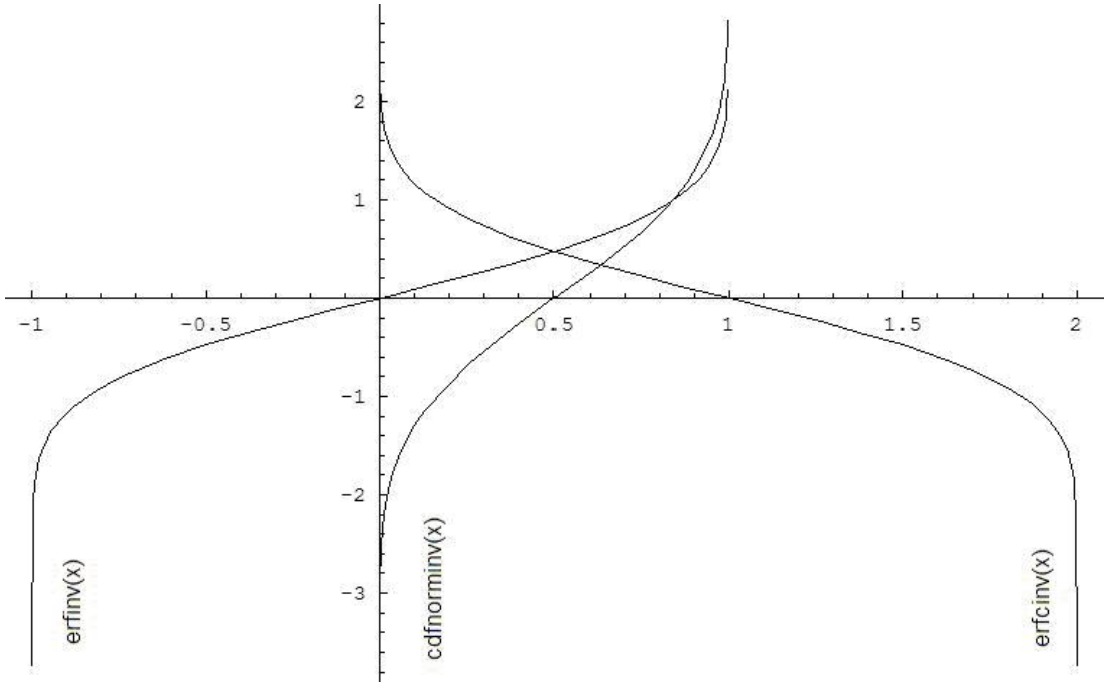
3. $\Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1) ,$

where $\Phi^{-1}(x)$ and $\operatorname{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\operatorname{erf}(x)$ respectively.

Figure "ErfInv Family Functions Relationship" illustrates the relationships among ErfInv family functions (ErfInv, ErfcInv, CdfNormInv).

__border__top

ErfInv Family Functions Relationship



Useful relations for these functions:

$\operatorname{erfcinv}(x) = \operatorname{erfinv}(1 - x)$

$\operatorname{cdfnorminv}(x) = \sqrt{2}\operatorname{erfinv}(2x - 1) = \sqrt{2}\operatorname{erfcinv}(2 - 2x)$

Special Values for Real Function v?ErfInv(x)

Argument	Result	VM Error Status	Exception
+0	+0		
-0	-0		
+1	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-1	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$ X > 1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[ErfcInv](#)

[CdfNormInv](#)

v?ErfcInv

Computes the inverse complementary error function value of vector elements.

Syntax

```

vsErfcInv( n, a, y );
vsErfcInvI(n, a, inca, y, incy);
vmsErfcInv( n, a, y, mode );
vmsErfcInvI(n, a, inca, y, incy, mode);
vdErfcInv( n, a, y );
vdErfcInvI(n, a, inca, y, incy);
vmdErfcInv( n, a, y, mode );
vmdErfcInvI(n, a, inca, y, incy, mode);

```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsErfcInv</code> , <code>vmsErfcInv</code> <code>const double*</code> for <code>vdErfcInv</code> , <code>vmdErfcInv</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsErfcInv</code> , <code>vmsErfcInv</code> <code>double*</code> for <code>vdErfcInv</code> , <code>vmdErfcInv</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `ErfcInv` function computes the inverse complimentary error function values for elements of the input vector *a* and writes them to the output vector *y*.

The inverse complimentary error function is defined as given by:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

$$\text{erfinv}(x) = \text{erf}^{-1}(x)$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

where `erf(x)` denotes the error function and `erfinv(x)` denotes the inverse error function.

See also [Figure "ErfInv Family Functions Relationship"](#) in `ErfInv` function description for `ErfcInv` function relationship with the other functions of `ErfInv` family.

Special Values for Real Function `v?ErfcInv(x)`

Argument	Result	VM Error Status	Exception
+1	+0		
+2	$-\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
$X < -0$	QNAN	VML_STATUS_ERRDOM	INVALID
$X > +2$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[ErfInv](#)

[CdfNormInv](#)

`v?CdfNormInv`

Computes the inverse cumulative normal distribution function values of vector elements.

Syntax

```

vsCdfNormInv( n, a, y );
vsCdfNormInvI(n, a, inca, y, incy);
vmsCdfNormInv( n, a, y, mode );
vmsCdfNormInvI(n, a, inca, y, incy, mode);
vdCdfNormInv( n, a, y );
vdCdfNormInvI(n, a, inca, y, incy);
vmdCdfNormInv( n, a, y, mode );
vmdCdfNormInvI(n, a, inca, y, incy, mode);

```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsCdfNormInv</code> , <code>vmsCdfNormInv</code> <code>const double*</code> for <code>vdCdfNormInv</code> , <code>vmdCdfNormInv</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsCdfNormInv</code> , <code>vmsCdfNormInv</code> <code>double*</code> for <code>vdCdfNormInv</code> , <code>vmdCdfNormInv</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `CdfNormInv` function computes the inverse cumulative normal distribution function values for elements of the input vector *a* and writes them to the output vector *y*.

The inverse cumulative normal distribution function is defined as given by:

$$\text{CdfNormInv}(x) = \text{CdfNorm}^{-1}(x) ,$$

where $\text{CdfNorm}(x)$ denotes the cumulative normal distribution function.

Useful relations:

$$\text{cdfnorminv}(x) = \sqrt{2}\text{erfinv}(2x - 1) = \sqrt{2}\text{erfcinv}(2 - 2x)$$

where $\text{erfinv}(x)$ denotes the inverse error function and $\text{erfcinv}(x)$ denotes the inverse complementary error functions.

See also [Figure "ErfInv Family Functions Relationship"](#) in [ErfInv](#) function description for [CdfNormInv](#) function relationship with the other functions of [ErfInv](#) family.

Special Values for Real Function $v\text{?CdfNormInv}(x)$

Argument	Result	VM Error Status	Exception
+0.5	+0		
+1	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
+0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$X < -0$	QNAN	VML_STATUS_ERRDOM	INVALID
$X > +1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[ErfInv](#)

[ErfcInv](#)

$v\text{?LGamma}$

Computes the natural logarithm of the absolute value of gamma function for vector elements.

Syntax

```
vsLGamma( n, a, y );
vsLGammaI(n, a, inca, y, incy);
vmsLGamma( n, a, y, mode );
vmsLGammaI(n, a, inca, y, incy, mode);
vdLGamma( n, a, y );
vdLGammaI(n, a, inca, y, incy);
vmdLGamma( n, a, y, mode );
vmdLGammaI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsLGamma</code> , <code>vmsLGamma</code> <code>const double*</code> for <code>vdLGamma</code> , <code>vmdLGamma</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsLGamma</code> , <code>vmsLGamma</code> <code>double*</code> for <code>vdLGamma</code> , <code>vmdLGamma</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?LGamma` function computes the natural logarithm of the absolute value of gamma function for elements of the input vector *a* and writes them to the output vector *y*. Precision overflow thresholds for the `v?LGamma` function are beyond the scope of this document. If the result does not meet the target precision, the function raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW`.

Special Values for Real Function `v?LGamma(x)`

Argument	Result	VM Error Status	Exception
+1	+0		
+2	+0		
+0	$+\infty$	<code>VML_STATUS_SING</code>	<code>ZERODIVIDE</code>
-0	$+\infty$	<code>VML_STATUS_SING</code>	<code>ZERODIVIDE</code>
negative integer	$+\infty$	<code>VML_STATUS_SING</code>	<code>ZERODIVIDE</code>
$-\infty$	$+\infty$		
$+\infty$	$+\infty$		
$X > \text{overflow}$	$+\infty$	<code>VML_STATUS_OVERFLOW</code>	<code>OVERFLOW</code>
QNaN	QNaN		
SNAN	QNaN		<code>INVALID</code>

`v?TGamma`

Computes the gamma function of vector elements.

Syntax

```
vsTGamma( n, a, y );
```

```

vsTGammaI(n, a, inca, y, incy);
vmsTGamma( n, a, y, mode );
vmsTGammaI(n, a, inca, y, incy, mode);
vdTGamma( n, a, y );
vdTGammaI(n, a, inca, y, incy);
vmdTGamma( n, a, y, mode );
vmdTGammaI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsTGamma, vmsTGamma const double* for vdTGamma, vmdTGamma	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsTGamma, vmsTGamma double* for vdTGamma, vmdTGamma	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?TGamma` function computes the gamma function for elements of the input vector *a* and writes them to the output vector *y*. Precision overflow thresholds for the `v?TGamma` function are beyond the scope of this document. If the result does not meet the target precision, the function raises the `OVERFLOW` exception and sets the VM Error Status to `VML_STATUS_OVERFLOW`.

Special Values for Real Function `v?TGamma(x)`

Argument	Result	VM Error Status	Exception
+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
negative integer	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
$X > \text{overflow}$	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW

Argument	Result	VM Error Status	Exception
QNAN	QNAN		
SNAN	QNAN		INVALID

v?ExpInt1

Computes the exponential integral of vector elements.

Syntax

```
vsExpInt1( n, a, y );
vsExpInt1I(n, a, inca, y, incy);
vmsExpInt1( n, a, y, mode );
vmsExpInt1I(n, a, inca, y, incy, mode);
vdExpInt1( n, a, y );
vdExpInt1I(n, a, inca, y, incy);
vmdExpInt1( n, a, y, mode );
vmdExpInt1I(n, a, inca, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsExpInt1, vmsExpInt1 const double* for vdExpInt1, vmdExpInt1	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsExpInt1, vmsExpInt1 double* for vdExpInt1, vmdExpInt1	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?ExpInt1` function computes the exponential integral E_1 of vector elements.

For positive real values x , this can be written as:

$$E_1(x) = \int_x^\infty \frac{e^{-t}}{t} dt = \int_1^\infty \frac{e^{-xt}}{t} dt.$$

For negative real values x , the result is defined as `NAN`.

Special Values for Real Function `v?ExpInt1(x)`

Argument	Result	VM Error Status	Exception
$x < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
$+0$	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
$+\infty$	$+0$		
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Rounding Functions

`v?Floor`

Computes an integer value rounded towards minus infinity for each vector element.

Syntax

```
vsFloor( n, a, y );
vsFloorI(n, a, inca, y, incy);
vmsFloor( n, a, y, mode );
vmsFloorI(n, a, inca, y, incy, mode);
vdFloor( n, a, y );
vdFloorI(n, a, inca, y, incy);
vmdFloor( n, a, y, mode );
vmdFloorI(n, a, inca, y, incy, mode);
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsFloor</code> , <code>vmsFloor</code> <code>const double*</code> for <code>vdFloor</code> , <code>vmdFloor</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .

Name	Type	Description
<code>mode</code>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<code>y</code>	<code>float*</code> for <code>vsFloor</code> , <code>vmsFloor</code> <code>double*</code> for <code>vdFloor</code> , <code>vmdFloor</code>	Pointer to an array that contains the output vector <code>y</code> .

Description

The function computes an integer value rounded towards minus infinity for each vector element.

$$y_i = \lfloor a_i \rfloor$$

Special Values for Real Function `v?Floor(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID
QNAN	QNAN	

`v?Ceil`

Computes an integer value rounded towards plus infinity for each vector element.

Syntax

```
vsCeil( n, a, y );
vsCeilI(n, a, inca, y, incy);
vmsCeil( n, a, y, mode );
vmsCeilI(n, a, inca, y, incy, mode);
vdCeil( n, a, y );
```

```
vdCeilI(n, a, inca, y, incy);
vmdCeil( n, a, y, mode );
vmdCeilI(n, a, inca, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

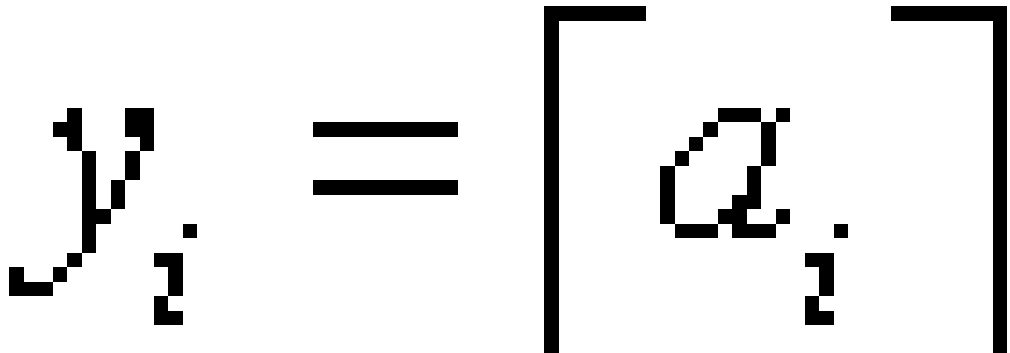
Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsCeil, vmsCeil const double* for vdCeil, vmdCeil	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsCeil, vmsCeil double* for vdCeil, vmdCeil	Pointer to an array that contains the output vector <i>y</i> .

Description

The function computes an integer value rounded towards plus infinity for each vector element.



Special Values for Real Function v?Ceil(x)

Argument	Result	Exception
+0	+0	
-0	-0	

Argument	Result	Exception
$+\infty$	$+\infty$	INVALID
$-\infty$	$-\infty$	
SNAN	QNAN	
QNAN	QNAN	

v?Trunc

Computes an integer value rounded towards zero for each vector element.

Syntax

```
vsTrunc( n, a, y );
vsTruncI(n, a, inca, y, incy);
vmsTrunc( n, a, y, mode );
vmsTruncI(n, a, inca, y, incy, mode);
vdTrunc( n, a, y );
vdTruncI(n, a, inca, y, incy);
vmdTrunc( n, a, y, mode );
vmdTruncI(n, a, inca, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsTrunc, vmsTrunc const double* for vdTrunc, vmdTrunc	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsTrunc, vmsTrunc double* for vdTrunc, vmdTrunc	Pointer to an array that contains the output vector <i>y</i> .

Description

The function computes an integer value rounded towards zero for each vector element.

$$a_i \geq 0, y_i = \lfloor a_i \rfloor$$

$$a_i < 0, y_i = \lceil a_i \rceil$$

Special Values for Real Function v?Trunc(x)

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Round

Computes a value rounded to the nearest integer for each vector element.

Syntax

```
vsRound( n, a, y );
vsRoundI(n, a, inca, y, incy);
vmsRound( n, a, y, mode );
vmsRoundI(n, a, inca, y, incy, mode);
vdRound( n, a, y );
vdRoundI(n, a, inca, y, incy);
vmdRound( n, a, y, mode );
vmdRoundI(n, a, inca, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsRound, vmsRound const double* for vdRound, vmdRound	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsRound, vmsRound double* for vdRound, vmdRound	Pointer to an array that contains the output vector <i>y</i> .

Description

The function computes a value rounded to the nearest integer for each vector element. Input elements that are halfway between two consecutive integers are always rounded away from zero regardless of the rounding mode.

Special Values for Real Function v?Round(x)

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?NearbyInt

Computes a rounded integer value in the current rounding mode for each vector element.

Syntax

```
vsNearbyInt( n, a, y );
vsNearbyIntI(n, a, inca, y, incy);
vmsNearbyInt( n, a, y, mode );
vmsNearbyIntI(n, a, inca, y, incy, mode);
vdNearbyInt( n, a, y );
vdNearbyIntI(n, a, inca, y, incy);
```

```
vmdNearbyInt( n, a, y, mode );
vmdNearbyIntI(n, a, inca, y, incy, mode);
```

Include Files

- `mkL.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsNearbyInt</code> , <code>vmsNearbyInt</code> <code>const double*</code> for <code>vdNearbyInt</code> , <code>vmdNearbyInt</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsNearbyInt</code> , <code>vmsNearbyInt</code> <code>double*</code> for <code>vdNearbyInt</code> , <code>vmdNearbyInt</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?NearbyInt` function computes a rounded integer value in a current rounding mode for each vector element.

Special Values for Real Function `v?NearbyInt(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID
QNAN	QNAN	

`v?Rint`

Computes a rounded integer value in the current rounding mode.

Syntax

```
vsRint( n, a, y );
```

```

vsRintI(n, a, inca, y, incy);
vmsRint( n, a, y, mode );
vmsRintI(n, a, inca, y, incy, mode);
vdRint( n, a, y );
vdRintI(n, a, inca, y, incy);
vmdRint( n, a, y, mode );
vmdRintI(n, a, inca, y, incy, mode);

```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsRint, vmsRint const double* for vdRint, vmdRint	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsRint, vmsRint double* for vdRint, vmdRint	Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Rint` function computes a rounded floating-point integer value using the current rounding mode for each vector element.

The rounding mode affects the results computed for inputs that fall between consecutive integers. For example:

- $f(0.5) = 0$, for rounding modes set to round to nearest round toward zero or to minus infinity.
- $f(0.5) = 1$, for rounding modes set to plus infinity.
- $f(-1.5) = -2$, for rounding modes set to round to nearest or to minus infinity.
- $f(-1.5) = -1$, for rounding modes set to round toward zero or to plus infinity.

Special Values for Real Function `v?Rint(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	

Argument	Result	Exception
$-\infty$	$-\infty$	INVALID
SNAN	QNAN	
QNAN	QNAN	

v?Modf

Computes a truncated integer value and the remaining fraction part for each vector element.

Syntax

```
vsModf( n, a, y, z );
vsModfI(n, a, inca, y, incy, z, incz);
vmsModf( n, a, y, z, mode );
vmsModfI(n, a, inca, y, incy, z, incz, mode);
vdModf( n, a, y, z );
vdModfI(n, a, inca, y, incy, z, incz);
vmdModf( n, a, y, z, mode );
vmdModfI(n, a, inca, y, incy, z, incz, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsModf, vmsModf const double* for vdModf, vmdModf	Pointer to an array that contains the input vector <i>a</i> .
<i>inca, incy, incz</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>y</i> , and <i>z</i> .
<i>mode</i>	const MKL_INT64	Overrides global VM mode setting for this function call. See vm1SetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y, z</i>	float* for vsModf, vmsModf double* for vdModf, vmdModf	Pointer to an array that contains the output vector <i>y</i> and <i>z</i> .

Description

The function computes a truncated integer value and the remaining fraction part for each vector element.

$$a_i \geq 0, \begin{cases} y_i = \lfloor a_i \rfloor \\ z_i = a_i - \lfloor a_i \rfloor \end{cases}$$

$$a_i < 0, \begin{cases} y_i = \lceil a_i \rceil \\ z_i = a_i - \lceil a_i \rceil \end{cases}$$

Special Values for Real Function v?Modf(x)

Argument	Result: $y(i)$	Result: $z(i)$	Exception
+0	+0	+0	INVALID
-0	-0	-0	
$+\infty$	$+\infty$	+0	
$-\infty$	$-\infty$	-0	
SNAN	QNAN	QNAN	
QNAN	QNAN	QNAN	

v?Frac

Computes a signed fractional part for each vector element.

Syntax

```
vsFrac( n, a, y );
vsFracI(n, a, inca, y, incy);
vmsFrac( n, a, y, mode );
vmsFracI(n, a, inca, y, incy, mode);
vdFrac( n, a, y );
vdFracI(n, a, inca, y, incy);
vmdFrac( n, a, y, mode );
vmdFracI(n, a, inca, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a</i>	<code>const float*</code> for <code>vsFrac</code> , <code>vmsFrac</code> <code>const double*</code> for <code>vdFrac</code> , <code>vmdFrac</code>	Pointer to an array that contains the input vector <i>a</i> .
<i>inca</i> , <i>incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsFrac</code> , <code>vmsFrac</code> <code>double*</code> for <code>vdFrac</code> , <code>vmdFrac</code>	Pointer to an array that contains the output vector <i>y</i> .

Description

The function computes a signed fractional part for each vector element.

$$y_i = \begin{cases} a_i - \lfloor a_i \rfloor, & a_i \geq 0 \\ a_i - \lceil a_i \rceil, & a_i < 0 \end{cases}$$

Special Values for Real Function `v?Frac(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+0	
-∞	-0	
SNAN	QNAN	INVALID
QNAN	QNAN	

VM Pack/Unpack Functions

This section describes VM functions that convert vectors with unit increment to and from vectors with positive increment indexing, vector indexing, and mask indexing (see Appendix "Vector Arguments in VM" for details on vector indexing methods).

The table below lists available VM Pack/Unpack functions, together with data types and indexing methods associated with them.

VM Pack/Unpack Functions

Function Short Name	Data Types	Indexing Methods	Description
v?Pack	s, d, c, z	I, V, M	Gathers elements of arrays, indexed by different methods.
v?Unpack	s, d, c, z	I, V, M	Scatters vector elements to arrays with different indexing.

See Also

Appendix "Vector Arguments in VM"

v?Pack

Copies elements of an array with specified indexing to a vector with unit increment.

Syntax

```
vsPackI( n, a, inca, y );
vsPackV( n, a, ia, y );
vsPackM( n, a, ma, y );
vdPackI( n, a, inca, y );
vdPackV( n, a, ia, y );
vdPackM( n, a, ma, y );
vcPackI( n, a, inca, y );
vcPackV( n, a, ia, y );
vcPackM( n, a, ma, y );
vzPackI( n, a, inca, y );
vzPackV( n, a, ia, y );
vzPackM( n, a, ma, y );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsPackI, vsPackV, vsPackM	Specifies pointer to an array that contains the input vector <i>a</i> . The arrays must be:

Name	Type	Description
	const double* for vdPackI, vdPackV, vdPackM	for v?PackI, at least $(1 + (n-1) * inca)$
	const MKL_Complex8* for vcPackI, vcPackV, vcPackM	for v?PackV, at least $\max(n, \max(ia[j]), j=0, \dots, n-1)$
	const MKL_Complex16* for vzPackI, vzPackV, vzPackM	for v?PackM, at least n .
<i>inca</i>	const MKL_INT for vsPackI, vdPackI, vcPackI, vzPackI	Specifies the increment for the elements of <i>a</i> .
<i>ia</i>	const int* for vsPackV, vdPackV, vcPackV, vzPackV	Specifies the pointer to an array of size at least n that contains the index vector for the elements of <i>a</i> .
<i>ma</i>	const int* for vsPackM, vdPackM, vcPackM, vzPackM	Specifies the pointer to an array of size at least n that contains the mask vector for the elements of <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsPackI, vsPackV, vsPackM	Pointer to an array of size at least n that contains the output vector <i>y</i> .
	double* for vdPackI, vdPackV, vdPackM	
	const MKL_Complex8* for vcPackI, vcPackV, vcPackM	
	const MKL_Complex16* for vzPackI, vzPackV, vzPackM	

v?Unpack

Copies elements of a vector with unit increment to an array with specified indexing.

Syntax

```

vsUnpackI( n, a, y, incy );
vsUnpackV( n, a, y, iy );
vsUnpackM( n, a, y, my );
vdUnpackI( n, a, y, incy );
vdUnpackV( n, a, y, iy );
vdUnpackM( n, a, y, my );
vcUnpackI( n, a, y, incy );
vcUnpackV( n, a, y, iy );
vcUnpackM( n, a, y, my );
vzUnpackI( n, a, y, incy );

```

```
vzUnpackV( n, a, y, iy );
```

```
vzUnpackM( n, a, y, my );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsUnpackI, vsUnpackV, vsUnpackM const double* for vdUnpackI, vdUnpackV, vdUnpackM const MKL_Complex8* for vcUnpackI, vcUnpackV, vcUnpackM const MKL_Complex16* for vzUnpackI, vzUnpackV, vzUnpackM	Specifies the pointer to an array of size at least <i>n</i> that contains the input vector <i>a</i> .
<i>incy</i>	const MKL_INT for vsUnpackI, vdUnpackI, vcUnpackI, vzUnpackI	Specifies the increment for the elements of <i>y</i> .
<i>iy</i>	const int* for vsUnpackV, vdUnpackV, vcUnpackV, vzUnpackV	Specifies the pointer to an array of size at least <i>n</i> that contains the index vector for the elements of <i>a</i> .
<i>my</i>	const int* for vsUnpackM, vdUnpackM, vcUnpackM, vzUnpackM	Specifies the pointer to an array of size at least <i>n</i> that contains the mask vector for the elements of <i>a</i> .

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsUnpackI, vsUnpackV, vsUnpackM double* for vdUnpackI, vdUnpackV, vdUnpackM const MKL_Complex8* for vcUnpackI, vcUnpackV, vcUnpackM const MKL_Complex16* for vzUnpackI, vzUnpackV, vzUnpackM	Specifies the pointer to an array that contains the output vector <i>y</i> . The array must be: for v?UnpackI, at least $(1 + (n-1) * incy)$ for v?UnpackV, at least $\max(n, \max(ia[j])), j=0, \dots, n-1,$ for v?UnpackM, at least <i>n</i> .

VM Service Functions

The VM Service functions enable you to set/get the accuracy mode and error code. These functions are available both in the Fortran and C interfaces. The table below lists available VM Service functions and their short description.

VM Service Functions

Function Short Name	Description
<code>vmlSetMode</code>	Sets the VM mode
<code>vmlGetMode</code>	Gets the VM mode
<code>MKLFreeTls</code>	Frees allocated VM/VS thread local storage memory from within DllMain routine (Windows* OS only)
<code>vmlSetErrStatus</code>	Sets the VM Error Status
<code>vmlGetErrStatus</code>	Gets the VM Error Status
<code>vmlClearErrStatus</code>	Clears the VM Error Status
<code>vmlSetErrorCallBack</code>	Sets the additional error handler callback function
<code>vmlGetErrorCallBack</code>	Gets the additional error handler callback function
<code>vmlClearErrorCallBack</code>	Deletes the additional error handler callback function

Product and Performance Information

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

Notice revision #20201201

vmlSetMode

Sets a new mode for VM functions according to the *mode* parameter and stores the previous VM mode to *oldmode*.

Syntax

```
oldmode = vmlSetMode( mode );
```

Include Files

- `mkh.h`

Input Parameters

Name	Type	Description
<i>mode</i>	<code>const MKL_UINT</code>	Specifies the VM mode to be set.

Output Parameters

Name	Type	Description
<i>oldmode</i>	<code>unsigned int</code>	Specifies the former VM mode.

Description

The `vmlSetMode` function sets a new mode for VM functions according to the *mode* parameter and stores the previous VM mode to *oldmode*. The mode change has a global effect on all the VM functions within a thread.

NOTE

You can override the global mode setting and change the mode for a given VM function call by using a respective `vm[s,d]<Func>` variant of the function.

The *mode* parameter is designed to control accuracy, handling of denormalized numbers, and error handling. Table "Values of the *mode* Parameter" lists values of the *mode* parameter. You can obtain all other possible values of the *mode* parameter from the *mode* parameter values by a using bitwise OR (|) operation to combine one value for accuracy, one value for handling of denormalized numbers, and one value for error control options. The default value of the *mode* parameter is `VML_HA | VML_FTZDAZ_CURRENT | VML_ERRMODE_DEFAULT`.

The `VML_FTZDAZ_ON` mode is specifically designed to improve the performance of computations that involve denormalized numbers at the cost of reasonable accuracy loss. This mode changes the numeric behavior of the functions: denormalized input values are treated as zeros (`DAZ` = denormals-are-zero) and denormalized results are flushed to zero (`FTZ` = flush-to-zero). Accuracy loss may occur if input and/or output values are close to denormal range.

Values of the *mode* Parameter

Value of <i>mode</i>	Description
Accuracy Control	
<code>VML_HA</code>	high accuracy versions of VM functions
<code>VML_LA</code>	low accuracy versions of VM functions
<code>VML_EP</code>	enhanced performance accuracy versions of VM functions
Denormalized Numbers Handling Control	
<code>VML_FTZDAZ_ON</code>	Faster processing of denormalized inputs is enabled.
<code>VML_FTZDAZ_OFF</code>	Faster processing of denormalized inputs is disabled.
<code>VML_FTZDAZ_CURRENT</code>	Keep the current CPU settings for denormalized inputs.
Error Mode Control	
<code>VML_ERRMODE_IGNORE</code>	On computation error, VM Error status is updated, but otherwise no action is set. Cannot be combined with other <code>VML_ERRMODE</code> settings.
<code>VML_ERRMODE_NOERR</code>	On computation error, VM Error status is not updated and no action is set. Cannot be combined with other <code>VML_ERRMODE</code> settings.
<code>VML_ERRMODE_ERRNO</code>	On error, the <i>errno</i> variable is set.
<code>VML_ERRMODE_STDERR</code>	On error, the error text information is written to <i>stderr</i> .
<code>VML_ERRMODE_EXCEPT</code>	On error, an exception is raised.
<code>VML_ERRMODE_CALLBACK</code>	On error, an additional error handler function is called.
<code>VML_ERRMODE_DEFAULT</code>	On error, the <i>errno</i> variable is set, an exception is raised, and an additional error handler function is called.

Product and Performance Information

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

Notice revision #20201201

Examples

The following example shows how to set low accuracy, fast processing for denormalized numbers and `stderr` error mode:

```
vmlSetMode( VML_LA );
vmlSetMode( VML_LA | VML_FTZDAZ_ON | VML_ERRMODE_STDERR );
```

vmlGetMode

Gets the VM mode.

Syntax

```
mod = vmlGetMode( void );
```

Include Files

- mkl.h

Output Parameters

Name	Type	Description
<i>mod</i>	unsigned int	Specifies the packed <i>mode</i> parameter.

Description

The function `vmlGetMode` returns the VM *mode* parameter that controls accuracy, handling of denormalized numbers, and error handling options. The *mod* variable value is a combination of the values listed in the table "Values of the *mode* Parameter". You can obtain these values using the respective mask from the table "Values of Mask for the *mode* Parameter".

Values of Mask for the *mode* Parameter

Value of mask	Description
VML_ACCURACY_MASK	Specifies mask for accuracy <i>mode</i> selection.
VML_FTZDAZ_MASK	Specifies mask for FTZDAZ <i>mode</i> selection.
VML_ERRMODE_MASK	Specifies mask for error <i>mode</i> selection.

See example below:

Examples

```
accm = vmlGetMode(void ) & VML_ACCURACY_MASK;
denm = vmlGetMode(void ) & VML_FTZDAZ_MASK;
errm = vmlGetMode(void ) & VML_ERRMODE_MASK;
```

MKLFreeTls

Frees allocated VM/VS thread local storage memory from within DllMain routine. Use on Windows OS only.*

Syntax

```
void MKLFreeTls( const MKL_UINT fdwReason );
```

Include Files

- mkl_vml_functions_win.h

Description

The `MKLFreeTls` routine frees thread local storage (TLS) memory which has been allocated when using MKL static libraries to link into a DLL on the Windows* OS. The routine should only be used within `DllMain`.

NOTE

It is only necessary to use `MKLFreeTls` for TLS data in the VM and VS domains.

Input Parameters

Name	Type	Description
<i>fdwReason</i>	<code>const MKL_UINT</code>	Reason code from the <code>DllMain</code> call.

Example

```

BOOL WINAPI DllMain(HINSTANCE hInst, DWORD fdwReason, LPVOID lpvReserved)
{
    /*customer code*/
    MKLFreeTls(fdwReason);
    /*customer code*/
}

```

vmlSetErrStatus

Sets the new VM Error Status according to `err` and stores the previous VM Error Status to `olderr`. Sets the global VM Status according to new values and returns the previous VM Status.

Syntax

```
olderr = vmlSetErrStatus( status );
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>status</i>	<code>const MKL_INT</code>	Specifies the VM error status to be set.

Output Parameters

Name	Type	Description
<i>olderr</i>	<code>int</code>	Specifies the former VM error status.

Description

Table "Values of the VM Status" lists possible values of the `err` parameter.

Values of the VM Status

Status	Description
--------	-------------

Successful Execution

<code>VML_STATUS_OK</code>	The execution was completed successfully.
----------------------------	---

Warnings

Status	Description
VML_STATUS_ACCURACYWARNING	The execution was completed successfully in a different accuracy mode.
Errors	
VML_STATUS_BADSIZE	The function does not support the preset accuracy mode. The Low Accuracy mode is used instead.
VML_STATUS_BADMEM	NULL pointer is passed.
VML_STATUS_ERRDOM	At least one of array values is out of a range of definition.
VML_STATUS_SING	At least one of the input array values causes a divide-by-zero exception or produces an invalid (QNaN) result.
VML_STATUS_OVERFLOW	An overflow has happened during the calculation process.
VML_STATUS_UNDERFLOW	An underflow has happened during the calculation process.

Examples

```
olderr = vmlSetErrStatus( VML_STATUS_OK );
```

```
olderr = vmlSetErrStatus( VML_STATUS_ERRDOM );
```

```
olderr = vmlSetErrStatus( VML_STATUS_UNDERFLOW );
```

vmlGetErrStatus

Gets the VM Error Status.

Syntax

```
err = vmlGetErrStatus( void );
```

Include Files

- mkl.h

Output Parameters

Name	Type	Description
<i>err</i>	int	Specifies the VM error status.

vmlClearErrStatus

Sets the VM Error Status to VML_STATUS_OK and stores the previous VM Error Status to olderr.

Syntax

```
olderr = vmlClearErrStatus( void );
```

Include Files

- mkl.h

Output Parameters

Name	Type	Description
<i>olderr</i>	int	Specifies the former VM error status.

vmlSetErrorCallBack

Sets the additional error handler callback function and gets the old callback function.

Syntax

```
oldcallback = vmlSetErrorCallBack( callback );
```

Include Files

- `mkl.h`

Input Parameters

Name	Description
<i>callback</i>	Pointer to the callback function.
The callback function has the following format:	
<pre>static int __cdecl MyHandler(DefVmlErrorContext* pContext) { /* Handler body */ };</pre>	

Name	Description
	The passed error structure is defined as follows:
	<pre>typedef struct _DefVmlErrorContext { int iCode; /* Error status value */ int iIndex; /* Index for bad array element, or bad array dimension, or bad array pointer */ double dbA1; /* Error argument 1 */ double dbA2; /* Error argument 2 */ double dbR1; /* Error result 1 */ double dbR2; /* Error result 2 */ char cFuncName[64]; /* Function name */ int iFuncNameLen; /* Length of functionname*/ double dbA1Im; /* Error argument 1, imag part*/ double dbA2Im; /* Error argument 2, imag part*/ double dbR1Im; /* Error result 1, imag part*/ double dbR2Im; /* Error result 2, imag part*/ } DefVmlErrorContext;</pre>

Output Parameters

Name	Type	Description
<i>oldcallback</i>	int	Pointer to the former callback function.

Description

The callback function is called on each VM mathematical function error if `VML_ERRMODE_CALLBACK` error mode is set (see "[Values of the *mode* Parameter](#)").

Use the `vmlSetErrorCallBack()` function if you need to define your own callback function instead of default empty callback function.

The input structure for a callback function contains the following information about the error encountered:

- the input value that caused an error
- location (array index) of this value
- the computed result value
- error code

- name of the function in which the error occurred.

You can insert your own error processing into the callback function. This may include correcting the passed result values in order to pass them back and resume computation. The standard error handler is called after the callback function only if it returns 0.

vmlGetErrorCallBack

Gets the additional error handler callback function.

Syntax

```
callback = vmlGetErrorCallBack( void );
```

Include Files

- mkl.h

Output Parameters

Name

callback

Description

Pointer to the callback function

vmlClearErrorCallBack

Deletes the additional error handler callback function and retrieves the former callback function.

Syntax

```
oldcallback = vmlClearErrorCallBack( void );
```

Include Files

- mkl.h

Output Parameters

Name

oldcallback

Type

int

Description

Pointer to the former callback function

Miscellaneous VM Functions

v?CopySign

Returns vector of elements of one argument with signs changed to match other argument elements.

Syntax

```
vsCopySign (n, a, y);
```

```
vsCopySignI(n, a, inca, b, incb, y, incy);
```

```
vmsCopySign (n, a, y, mode);
```

```
vmsCopySignI(n, a, inca, b, incb, y, incy, mode);
```

```
vdCopySign (n, a, y);
```

```
vdCopySignI(n, a, inca, b, incb, y, incy);
vmdCopySign (n, a, y, mode);
vmdCopySignI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	const float* for vsCopySign const float* for vmsCopySign const double* for vdCopySign const double* for vmdCopySign	Pointer to the array containing the input vector <i>a</i> .
<i>inca, incb, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsCopySign float* for vmsCopySign double* for vdCopySign double* for vmdCopySign	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?CopySign` function returns the first vector argument elements with the sign changed to match the sign of the second vector argument's corresponding elements.

v?NextAfter

Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector.

Syntax

```
vsNextAfter (n, a, b, y);
vsNextAfterI(n, a, inca, b, incb, y, incy);
vmsNextAfter (n, a, b, y, mode);
```

```

vmsNextAfterI(n, a, inca, b, incb, y, incy, mode);
vdNextAfter (n, a, b, y);
vdNextAfterI(n, a, inca, b, incb, y, incy);
vmdNextAfter (n, a, b, y, mode);
vmdNextAfterI(n, a, inca, b, incb, y, incy, mode);

```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a, b</i>	<code>const float*</code> for <code>vsNextAfter</code> <code>const float*</code> for <code>vmsNextAfter</code> <code>const double*</code> for <code>vdNextAfter</code> <code>const double*</code> for <code>vmdNextAfter</code>	Pointers to the arrays containing the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsNextAfter</code> <code>float*</code> for <code>vmsNextAfter</code> <code>double*</code> for <code>vdNextAfter</code> <code>double*</code> for <code>vmdNextAfter</code>	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?NextAfter` function returns a vector containing the next representable floating-point values following the first vector argument elements in the direction of the second vector argument's corresponding elements.

Special cases:

Overflow	The function raises overflow and inexact floating-point exceptions and sets <code>VML_STATUS_OVERFLOW</code> if an input vector argument element is finite and the corresponding result vector element value is infinite.
Underflow	The function raises underflow and inexact floating-point exceptions and sets <code>VML_STATUS_UNDERFLOW</code> if a result vector element value is subnormal or zero, and different from the corresponding input vector argument element.

Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

v?Fdim

Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise.

Syntax

```
vsFdim (n, a, b, y);
vsFdimI(n, a, inca, b, incb, y, incy);
vmsFdim (n, a, b, y, mode);
vmsFdimI(n, a, inca, b, incb, y, incy, mode);
vdFdim (n, a, b, y);
vdFdimI(n, a, inca, b, incb, y, incy);
vmdFdim (n, a, b, y, mode);
vmdFdimI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	const float* for vsFdim const float* for vmsFdim const double* for vdFdim const double* for vmdFdim	Pointers to the arrays containing the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsFdim float* for vmsFdim double* for vdFdim double* for vmdFdim	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?Fdim` function returns a vector containing the differences of the corresponding elements of the first and second vector arguments if the first element is larger, and +0 otherwise.

Special values for Real Function `v?Fdim(x, y)`

Argument 1	Argument 2	Result	VM Error Status	Exception
any	QNAN	QNAN		
any	SNAN	QNAN		INVALID
QNAN	any	QNAN		
SNAN	any	QNAN		INVALID

`v?Fmax`

Returns the larger of each pair of elements of the two vector arguments.

Syntax

```
vsFmax (n, a, b, y);
vsFmaxI(n, a, inca, b, incb, y, incy);
vmsFmax (n, a, b, y, mode);
vmsFmaxI(n, a, inca, b, incb, y, incy, mode);
vdFmax (n, a, b, y);
vdFmaxI(n, a, inca, b, incb, y, incy);
vmdFmax (n, a, b, y, mode);
vmdFmaxI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a, b</i>	<code>const float*</code> for <code>vsFmax</code> <code>const float*</code> for <code>vmsFmax</code> <code>const double*</code> for <code>vdFmax</code> <code>const double*</code> for <code>vmdFmax</code>	Pointers to the arrays containing the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	<code>float*</code> for <code>vsFmax</code>	Pointer to an array containing the output vector y .
	<code>float*</code> for <code>vmsFmax</code>	
	<code>double*</code> for <code>vdFmax</code>	
	<code>double*</code> for <code>vmdFmax</code>	

Description

The `v?Fmax` function returns a vector with element values equal to the larger value from each pair of corresponding elements of the two vectors a and b : if $a_i < b_i$, `v?Fmax` returns b_i , otherwise `v?Fmax` returns a_i .

Special values for Real Function `v?Fmax(x, y)`

Argument 1	Argument 2	Result	VM Error Status	Exception
a_i not NAN	NAN	a_i		
NAN	b_i not NAN	b_i		
NAN	NAN	NAN		

See Also

[Fmin](#) Returns the smaller of each pair of elements of the two vector arguments.

[MaxMag](#) Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

`v?Fmin`

Returns the smaller of each pair of elements of the two vector arguments.

Syntax

```
vsFmin (n, a, b, y);
vsFminI(n, a, inca, b, incb, y, incy);
vmsFmin (n, a, b, y, mode);
vmsFminI(n, a, inca, b, incb, y, incy, mode);
vdFmin (n, a, b, y);
vdFminI(n, a, inca, b, incb, y, incy);
vmdFmin (n, a, b, y, mode);
vmdFminI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
n	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.

Name	Type	Description
a, b	const float* for vsFmin const float* for vmsFmin const double* for vdFmin const double* for vmdFmin	Pointers to the arrays containing the input vectors a and b .
$inca, incb, incy$	const MKL_INT	Specifies increments for the elements of a , b , and y .
$mode$	const MKL_INT64	Overrides the global VM $mode$ setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	float* for vsFmin float* for vmsFmin double* for vdFmin double* for vmdFmin	Pointer to an array containing the output vector y .

Description

The `v?Fmin` function returns a vector with element values equal to the smaller value from each pair of corresponding elements of the two vectors a and b : if $b_i < a_i$, `v?Fmin` returns b_i , otherwise `v?Fmin` returns a_i .

Special values for Real Function `v?Fmin(x, y)`

Argument 1	Argument 2	Result	VM Error Status	Exception
a_i not NAN	NAN	a_i		
NAN	b_i not NAN	b_i		
NAN	NAN	NAN		

See Also

[Fmax](#) Returns the larger of each pair of elements of the two vector arguments.

[MinMag](#) Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

`v?MaxMag`

Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

Syntax

```
vsMaxMag (n, a, b, y);
vsMaxMagI(n, a, inca, b, incb, y, incy);
vmsMaxMag (n, a, b, y, mode);
vmsMaxMagI(n, a, inca, b, incb, y, incy, mode);
vdMaxMag (n, a, b, y);
```



```
vdMaxMagI(n, a, inca, b, incb, y, incy);
vmdMaxMag (n, a, b, y, mode);
vmdMaxMagI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>const MKL_INT</code>	Specifies the number of elements to be calculated.
<i>a, b</i>	<code>const float*</code> for <code>vsMaxMag</code> <code>const float*</code> for <code>vmsMaxMag</code> <code>const double*</code> for <code>vdMaxMag</code> <code>const double*</code> for <code>vmdMaxMag</code>	Pointers to the arrays containing the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	<code>const MKL_INT</code>	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	<code>const MKL_INT64</code>	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<code>float*</code> for <code>vsMaxMag</code> <code>float*</code> for <code>vmsMaxMag</code> <code>double*</code> for <code>vdMaxMag</code> <code>double*</code> for <code>vmdMaxMag</code>	Pointer to an array containing the output vector <i>y</i> .

Description

The `v?MaxMag` function returns a vector with element values equal to the element with the larger magnitude from each pair of corresponding elements of the two vectors *a* and *b*:

- If $|a_i| > |b_i|$ `v?MaxMag` returns a_i , otherwise `v?MaxMag` returns a_i .
- If $|b_i| > |a_i|$ `v?MaxMag` returns b_i , otherwise `v?MaxMag` returns a_i .
- Otherwise `v?MaxMag` behaves like `v?Fmax`.

Special values for Real Function `v?MaxMag(x, y)`

Argument 1	Argument 2	Result	VM Error Status	Exception
a_i not NAN	NAN	a_i		
NAN	b_i not NAN	b_i		
NAN	NAN	NAN		

See Also

MinMag Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

Fmax Returns the larger of each pair of elements of the two vector arguments.

v?MinMag

Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

Syntax

```
vsMinMag (n, a, b, y);
vsMinMagI(n, a, inca, b, incb, y, incy);
vmsMinMag (n, a, b, y, mode);
vmsMinMagI(n, a, inca, b, incb, y, incy, mode);
vdMinMag (n, a, b, y);
vdMinMagI(n, a, inca, b, incb, y, incy);
vmdMinMag (n, a, b, y, mode);
vmdMinMagI(n, a, inca, b, incb, y, incy, mode);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	const float* for vsMinMag const float* for vmsMinMag const double* for vdMinMag const double* for vmdMinMag	Pointers to the arrays containing the input vectors <i>a</i> and <i>b</i> .
<i>inca, incb, incy</i>	const MKL_INT	Specifies increments for the elements of <i>a</i> , <i>b</i> , and <i>y</i> .
<i>mode</i>	const MKL_INT64	Overrides the global VM <i>mode</i> setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	float* for vsMinMag float* for vmsMinMag double* for vdMinMag	Pointer to an array containing the output vector <i>y</i> .

Name	Type	Description
------	------	-------------

	double* for vmdMinMag	
--	-----------------------	--

Description

The `v?MinMag` function returns a vector with element values equal to the element with the smaller magnitude from each pair of corresponding elements of the two vectors a and b :

- If $|a_i| < |b_i|$ `v?MaxMag` returns a_i , otherwise `v?MaxMag` returns b_i .
- If $|b_i| < |a_i|$ `v?MaxMag` returns b_i , otherwise `v?MaxMag` returns a_i .
- Otherwise `v?MaxMag` behaves like `v?Fmin`.

Special values for Real Function `v?MinMag(x, y)`

Argument 1	Argument 2	Result	VM Error Status	Exception
a_i not NAN	NAN	a_i		
NAN	b_i not NAN	b_i		
NAN	NAN	NAN		

See Also

[MaxMag](#) Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

[Fmin](#) Returns the smaller of each pair of elements of the two vector arguments.

Statistical Functions

Statistical functions in Intel® oneAPI Math Kernel Library (oneMKL) are known as the Vector Statistics (VS). They are designed for the purpose of

- generating vectors of pseudorandom, quasi-random, and non-deterministic random numbers
- performing mathematical operations of convolution and correlation
- computing basic statistical estimates for single and double precision multi-dimensional datasets

The corresponding functionality is described in the respective [Random Number Generators](#), [Convolution and Correlation](#), and [Summary Statistics](#) topics.

See VS performance data in the online VS Performance Data document available at <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html>.

The basic notion in VS is a task. The task object is a data structure or descriptor that holds the parameters related to a specific statistical operation: random number generation, convolution and correlation, or summary statistics estimation. Such parameters can be an identifier of a random number generator, its internal state and parameters, data arrays, their shape and dimensions, an identifier of the operation and so forth. You can modify the VS task parameters using the VS service functions.

Product and Performance Information

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

Notice revision #20201201

Random Number Generators

Intel® oneAPI Math Kernel Library (oneMKL) VS provides a set of routines implementing commonly used pseudorandom, quasi-random, or non-deterministic random number generators with continuous and discrete distribution. To improve performance, all these routines were developed using the calls to the highly optimized *Basic Random Number Generators* (BRNGs) and vector mathematical functions (VM, see "[Vector Mathematical Functions](#)").

VS provides interfaces both for Fortran and C languages. For users of the C and C++ languages the `mkl_vs1.h` header file is provided. All header files are found in the following directory:

```
${MKL}/include
```

All VS routines can be classified into three major categories:

- Transformation routines for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These routines indirectly call basic random number generators, which are pseudorandom, quasi-random, or non-deterministic random number generators. Detailed description of the generators can be found in [Distribution Generators](#).
- Service routines to handle random number streams: create, initialize, delete, copy, save to a binary file, load from a binary file, get the index of a basic generator. The description of these routines can be found in [Service Routines](#).
- Registration routines for basic pseudorandom generators and routines that obtain properties of the registered generators (see [Advanced Service Routines](#)).

The last two categories are referred to as service routines.

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

Random Number Generators Conventions

This document makes no specific differentiation between random, pseudorandom, and quasi-random numbers, nor between random, pseudorandom, and quasi-random number generators unless the context requires otherwise. For details, refer to the '*Random Numbers*' section in [VS Notes](#) document provided at the Intel® oneAPI Math Kernel Library (oneMKL) web page.

All generators of nonuniform distributions, both discrete and continuous, are built on the basis of the uniform distribution generators, called Basic Random Number Generators (BRNGs). The pseudorandom numbers with nonuniform distribution are obtained through an appropriate transformation of the uniformly distributed pseudorandom numbers. Such transformations are referred to as *generation methods*. For a given distribution, several generation methods can be used. See [VS Notes](#) for the description of methods available for each generator.

An RNG task determines environment in which random number generation is performed, in particular parameters of the BRNG and its internal state. Output of VS generators is a stream of random numbers that are used in Monte Carlo simulations. A *random stream descriptor* and a *random stream* are used as synonyms of an *RNG task* in the document unless the context requires otherwise.

The *random stream descriptor* specifies which BRNG should be used in a given transformation method. See the *Random Streams and RNGs in Parallel Computation* section of [VS Notes](#).

The term *computational node* means a logical or physical unit that can process data in parallel.

Random Number Generators Mathematical Notation

The following notation is used throughout the text:

N	The set of natural numbers $N = \{1, 2, 3 \dots\}$.
Z	The set of integers $Z = \{\dots -3, -2, -1, 0, 1, 2, 3 \dots\}$.
R	The set of real numbers.

$$\lfloor a \rfloor$$

The floor of a (the largest integer less than or equal to a).

\oplus or **xor**

Bitwise exclusive OR.

$$C_{\alpha}^{\kappa} \text{ or } \binom{\alpha}{\kappa}$$

Binomial coefficient or combination ($\alpha \in R, \alpha \geq 0; \kappa \in \mathbb{N} \cup \{0\}$).

$$C_{\alpha}^0 = 1$$

For $\alpha \geq k$ binomial coefficient is defined as

$$C_{\alpha}^{\kappa} = \frac{\alpha(\alpha - 1) \dots (\alpha - \kappa + 1)}{\kappa!}$$

If $\alpha < k$, then

$$C_{\alpha}^{\kappa} = 0$$

$\Phi(x)$

Cumulative Gaussian distribution function

$$\Phi(x) = \int_{-\infty}^x \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) dy$$

defined over $-\infty < x < +\infty$.

$\Phi(-\infty) = 0, \Phi(+\infty) = 1$.

$\Gamma(\alpha)$

The complete gamma function

$$\Gamma(\alpha) = \int_0^{\infty} t^{\alpha-1} e^{-t} dt$$

where $\alpha > 0$.

$B(p, q)$

The complete beta function

$$B(p, q) = \int_0^1 t^{p-1} (1 - t)^{q-1} dt$$

where $p > 0$ and $q > 0$.

$LCG(a, c, m)$

Linear Congruential Generator $x_{n+1} = (ax_n + c) \bmod m$, where a is called the *multiplier*, c is called the *increment*, and m is called the *modulus* of the generator.

$MCG(a, m)$

Multiplicative Congruential Generator $x_{n+1} = (ax_n) \bmod m$ is a special case of Linear Congruential Generator, where the increment c is taken to be 0.

$GFSR(p, q)$

Generalized Feedback Shift Register Generator

$$x_n = x_{n-p} \oplus x_{n-q}.$$

Random Number Generators Naming Conventions

The names of the routines, types, and constants in VS random number generators are case-sensitive and can contain lowercase and uppercase characters (`viRngUniform`).

The names of generator routines have the following structure:

`v<type of result>Rng<distribution>`

where

- `v` is the prefix of a VS vector function.
- `<type of result>` is either `s`, `d`, or `i` and specifies one of the following types:

<code>s</code>	float
<code>d</code>	double
<code>i</code>	int

Prefixes `s` and `d` apply to continuous distributions only, prefix `i` applies only to discrete case.

- `rng` indicates that the routine is a random generator.
- `<distribution>` specifies the type of statistical distribution.

On 64-bit platforms, routines with the `_64` suffix support large data arrays in the LP64 interface library and enable you to mix integer types in one application. For more interface library details, see "Using the ILP64 Interface vs. LP64 Interface" in the developer guide.

Names of service routines follow the template below:

`vs1<name>`

where

- `vs1` is the prefix of a VS service function.
- `<name>` contains a short function name.

For a more detailed description of service routines, refer to [Service Routines](#) and [Advanced Service Routines](#).

The prototype of each generator routine corresponding to a given probability distribution fits the following structure:

```
status = <function name>( method, stream, n, r, [<distribution parameters>] )
```

where

- *method* defines the method of generation. A detailed description of this parameter can be found in table "Values of <method> in *method* parameter". See below, where the structure of the *method* parameter name is explained.
- *stream* defines the descriptor of the random stream and must have a non-zero value. Random streams, descriptors, and their usage are discussed further in [Random Streams](#) and [Service Routines](#).
- *n* defines the number of random values to be generated. If *n* is less than or equal to zero, no values are generated. Furthermore, if *n* is negative, an error condition is set.
- *r* defines the destination array for the generated numbers. The dimension of the array must be large enough to store at least *n* random numbers.
- *status* defines the error status of a VS routine. See [Error Reporting](#) for a detailed description of error status values.

Additional parameters included into <distribution parameters> field are individual for each generator routine and are described in detail in [Distribution Generators](#).

To invoke a distribution generator, use a call to the respective VS routine. For example, to obtain a vector *r*, composed of *n* independent and identically distributed random numbers with normal (Gaussian) distribution, that have the mean value *a* and standard deviation *sigma*, write the following:

```
status = vsRngGaussian( method, stream, n, r, a, sigma )
```

The name of a *method* parameter has the following structure:

```
VSL_RNG_METHOD_method<distribution>_<method>
```

```
VSL_RNG_METHOD_<distribution>_<method>_ACCURATE
```

where

- <distribution> is the probability distribution.
- <method> is the method name.

Type of the name structure for the *method* parameter corresponds to fast and accurate modes of random number generation (see "[Distribution Generators](#)" and [VS Notes](#) for details).

Method names VSL_RNG_METHOD_<distribution>_<method>

and

VSL_RNG_METHOD_<distribution>_<method>_ACCURATE

should be used with

```
v<precision>Rng<distribution>
```

function only, where

- <precision> is

<i>s</i>	for single precision continuous distribution
<i>d</i>	for double precision continuous distribution
<i>i</i>	for discrete distribution
- <distribution> is the probability distribution.

is the probability distribution. Table "Values of <method> in *method* parameter" provides specific predefined values of the *method* name. The third column contains names of the functions that use the given method.

Values of `<method>` in `method` parameter

Method	Short Description	Functions
STD	Standard method. Currently there is only one method for these functions.	Uniform (continuous), Uniform (discrete), UniformBits , UniformBits32 , UniformBits64
BOXMULLER	BOXMULLER generates normally distributed random number x thru the pair of uniformly distributed numbers u_1 and u_2 according to the formula: $x = \sqrt{-2 \ln u_1} \sin 2\pi u_2$	Gaussian , GaussianMV
BOXMULLER2	BOXMULLER2 generates normally distributed random numbers x_1 and x_2 thru the pair of uniformly distributed numbers u_1 and u_2 according to the formulas: $x_1 = \sqrt{-2 \ln u_1} \sin 2\pi u_2$ $x_2 = \sqrt{-2 \ln u_1} \cos 2\pi u_2$	Gaussian , GaussianMV , Lognormal
ICDF	Inverse cumulative distribution function method.	Exponential , Laplace , Weibull , Cauchy , Rayleigh , Gumbel , Bernoulli , Geometric , Gaussian , GaussianMV , Lognormal
GNORM	For $\alpha > 1$, a gamma distributed random number is generated as a cube of properly scaled normal random number; for $0.6 \leq \alpha < 1$, a gamma distributed random number is generated using rejection from Weibull distribution; for $\alpha < 0.6$, a gamma distributed random number is obtained using transformation of exponential power distribution; for $\alpha = 1$, gamma distribution is reduced to exponential distribution.	Gamma
CJA	For $\min(p, q) > 1$, Cheng method is used; for $\min(p, q) < 1$, Johnk method is used, if $q + K \cdot p^2 + C \leq 0$ ($K = 0.852...$, $C = -0.956...$) otherwise, Atkinson switching algorithm is used; for $\max(p, q) < 1$, method of Johnk is used; for $\min(p, q) < 1$, $\max(p, q) > 1$, Atkinson switching algorithm is used (CJA stands for the first letters of Cheng, Johnk, Atkinson); for $p = 1$ or $q = 1$, inverse cumulative distribution function method is used; for $p = 1$ and $q = 1$, beta distribution is reduced to uniform distribution.	Beta

Method	Short Description	Functions
BTPE	Acceptance/rejection method for $n_{trial} \cdot \min(p, 1 - p) \geq 30$ with decomposition into 4 regions: <ul style="list-style-type: none"> – 2 parallelograms – triangle – left exponential tail – right exponential tail 	Binomial
H2PE	Acceptance/rejection method for large mode of distribution with decomposition into 3 regions: <ul style="list-style-type: none"> – rectangular – left exponential tail – right exponential tail 	Hypergeometric
PTPE	Acceptance/rejection method for $\lambda \geq 27$ with decomposition into 4 regions: <ul style="list-style-type: none"> – 2 parallelograms – triangle – left exponential tail – right exponential tail; otherwise, table lookup method is used.	Poisson
POISNORM	for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$, table lookup method is used.	Poisson , PoissonV
NBAR	Acceptance/rejection method for , $\frac{(a - 1) \cdot (1 - p)}{p} \geq 100$ with decomposition into 5 regions: <ul style="list-style-type: none"> – rectangular – 2 trapezoid – left exponential tail – right exponential tail 	NegBinomial
CHI2GAMMA	Random number generator of chi-square distribution with ν degrees of freedom. To generate any successive random number x of the chi-square distribution: <ul style="list-style-type: none"> • If ν is 1 or 3, a chi-square distributed random number is generated as a sum of squares of ν independent normal random numbers with mean value $a = 0$ and standard deviation $\sigma = 1$. • If ν is even and $2 \leq \nu \leq 16$, a chi-square distributed random number is generated using the formula: $x = -2 \ln \left(\prod_{i=1}^{\nu/2} u_i \right)$ 	ChiSquare

Method	Short Description	Functions
	<p>where u_i are successive random numbers uniformly distributed over the interval (0, 1)</p> <ul style="list-style-type: none"> If $v \geq 17$ or v is odd and $5 \leq v \leq 15$, a chi-square distribution is reduced to a Gamma distribution with these parameters: <ul style="list-style-type: none"> Shape $a = v / 2$ Offset $a = 0$ Scale factor $\beta = 2$ <p>The random numbers of the Gamma distribution are generated using the VSL_RNG_METHOD_GAMMA_GNORM method.</p>	

NOTE

In this document, routines are often referred to by their base name ([Gaussian](#)) when this does not lead to ambiguity. In the routine reference, the full name ([vsrnggaussian](#), [vsRngGaussian](#)) is always used in prototypes and code examples.

Basic Generators

VS provides pseudorandom, quasi-random, and non-deterministic random number generators. This includes the following BRNGs, which differ in speed and other properties:

- the 31-bit multiplicative congruential pseudorandom number generator [MCG\(1132489760, \$2^{31}-1\$ \)](#) [[L'Ecuyer99](#)]
- the 32-bit generalized feedback shift register pseudorandom number generator [GFSR\(250, 103\)](#) [[Kirkpatrick81](#)]
- the combined multiple recursive pseudorandom number generator [MRG32k3a](#) [[L'Ecuyer99a](#)]
- the 59-bit multiplicative congruential pseudorandom number generator [MCG\(13¹³, \$2^{59}\$ \)](#) from NAG Numerical Libraries [[NAG](#)]
- Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [[NAG](#)]
- Mersenne Twister pseudorandom number generator [MT19937](#) [[Matsumoto98](#)] with period length $2^{19937}-1$ of the produced sequence
- Set of 6024 Mersenne Twister pseudorandom number generators [MT2203](#) [[Matsumoto98](#)], [[Matsumoto00](#)]. Each of them generates a sequence of period length equal to $2^{2203}-1$. Parameters of the generators provide mutual independence of the corresponding sequences.
- SIMD-oriented Fast Mersenne Twister pseudorandom number generator [SFMT19937](#) [[Saito08](#)] with a period length equal to $2^{19937}-1$ of the produced sequence.
- Sobol quasi-random number generator [[Sobol76](#)], [[Bratley88](#)], which works in arbitrary dimension. For dimensions greater than 40 the user should supply initialization parameters (initial direction numbers and primitive polynomials or direction numbers) by using [vslNewStreamEx](#) function. See additional details on interface for registration of the parameters in the library in [VS Notes](#).
- Niederreiter quasi-random number generator [[Bratley92](#)], which works in arbitrary dimension. For dimensions greater than 318 the user should supply initialization parameters (irreducible polynomials or direction numbers) by using [vslNewStreamEx](#) function. See additional details on interface for registration of the parameters in the library in [VS Notes](#).
- Non-deterministic random number generator (RDRAND-based generators only) [[AVX](#)], [[IntelSWMan](#)].

NOTE

You can use a non-deterministic random number generator only if the underlying hardware supports it. For instructions on how to detect if an Intel CPU supports a non-deterministic random number generator see, for example, [Chapter 8: Post-32nm Processor Instructions](#) in [[AVX](#)] or [Chapter 4: RdRand Instruction Usage](#) in [[BMT](#)].

NOTE

The time required by some non-deterministic sources to generate a random number is not constant, so you might have to make multiple requests before the next random number is available. VS limits the number of retries for requests to the non-deterministic source to 10. You can redefine the maximum number of retries during the initialization of the non-deterministic random number generator with the `vslNewStreamEx` function.

For more details on the non-deterministic source implementation for Intel CPUs please refer to Section 7.3.17, Volume 1, *Random Number Generator Instruction* in [IntelSWMan] and Section 4.2.2, *RdRand Retry Loop* in [BMT].

- Philox4x32-10 counter-based pseudorandom number generator with a period of 2^{128} `PHILOX4X32X10` [Salmon11].
- ARS-5 counter-based pseudorandom number generator with a period of 2^{128} , which uses instructions from the AES-NI set `ARS5` [Salmon11].

See some testing results for the generators in [VS Notes](#) and comparative performance data at <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html>.

VS provides means of registration of such user-designed generators through the steps described in [Advanced Service Routines](#).

For some basic generators, VS provides two methods of creating independent random streams in multiprocessor computations, which are the leapfrog method and the block-splitting method. These sequence splitting methods are also useful in sequential Monte Carlo.

In addition, MT2203 pseudorandom number generator is a set of 6024 generators designed to create up to 6024 independent random sequences, which might be used in parallel Monte Carlo simulations. Another generator that has the same feature is Wichmann-Hill. It allows creating up to 273 independent random streams. The properties of the generators designed for parallel computations are discussed in detail in [Coddington94].

You may want to design and use your own basic generators. VS provides means of registration of such user-designed generators through the steps described in [Advanced Service Routines](#).

There is also an option to utilize externally generated random numbers in VS distribution generator routines. For this purpose VS provides three additional basic random number generators:

- for external random data packed in 32-bit integer array
- for external random data stored in double precision floating-point array; data is supposed to be uniformly distributed over (a,b) interval
- for external random data stored in single precision floating-point array; data is supposed to be uniformly distributed over (a,b) interval.

Such basic generators are called the abstract basic random number generators.

See [VS Notes](#) for a more detailed description of the generator properties.

Product and Performance Information

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

Notice revision #20201201

BRNG Parameter Definition

Predefined values for the `brng` input parameter are as follows:

Values of `brng` parameter

Value	Short Description
<code>VSL_BRNG_MCG31</code>	A 31-bit multiplicative congruential generator.

Value	Short Description
VSL_BRNG_R250	A generalized feedback shift register generator.
VSL_BRNG_MRG32K3A	A combined multiple recursive generator with two components of order 3.
VSL_BRNG_MCG59	A 59-bit multiplicative congruential generator.
VSL_BRNG_WH	A set of 273 Wichmann-Hill combined multiplicative congruential generators.
VSL_BRNG_MT19937	A Mersenne Twister pseudorandom number generator.
VSL_BRNG_MT2203	A set of 6024 Mersenne Twister pseudorandom number generators.
VSL_BRNG_SFMT19937	A SIMD-oriented Fast Mersenne Twister pseudorandom number generator.
VSL_BRNG_SOBOL	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \leq s \leq 40$; user-defined dimensions are also available.
VSL_BRNG_NIEDERR	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \leq s \leq 318$; user-defined dimensions are also available.
VSL_BRNG_IABSTRACT	An abstract random number generator for integer arrays.
VSL_BRNG_DABSTRACT	An abstract random number generator for double precision floating-point arrays.
VSL_BRNG_SABSTRACT	An abstract random number generator for single precision floating-point arrays.
VSL_BRNG_NONDETERM	A non-deterministic random number generator.
VSL_BRNG_PHILOX4X32X10	A Philox4x32-10 counter-based pseudorandom number generator.
VSL_BRNG_ARS5	An ARS-5 counter-based pseudorandom number generator that uses instructions from the AES-NI set.

See [VS Notes](#) for detailed description.

Product and Performance Information

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

Notice revision #20201201

Random Streams

Random stream (or *stream*) is an abstract source of pseudo- and quasi-random sequences of uniform distribution. You can operate with stream state descriptors only. A stream state descriptor, which holds state descriptive information for a particular BRNG, is a necessary parameter in each routine of a distribution generator. Only the distribution generator routines operate with random streams directly. See [VS Notes](#) for details.

NOTE

Random streams associated with abstract basic random number generator are called the abstract random streams. See [VS Notes](#) for detailed description of abstract streams and their use.

You can create unlimited number of random streams by VS [Service Routines](#) like [NewStream](#) and utilize them in any distribution generator to get the sequence of numbers of given probability distribution. When they are no longer needed, the streams should be deleted calling service routine [DeleteStream](#).

VS provides service functions [SaveStreamF](#) and [LoadStreamF](#) to save random stream descriptive data to a binary file and to read this data from a binary file respectively. See [VS Notes](#) for detailed description.

BRNG Data Types

```
typedef (void*) VSLStreamStatePtr;
```

See [Advanced Service Routines](#) for the format of the stream state structure for user-designed generators.

Error Reporting

VS RNG routines return status codes of the performed operation to report errors to the calling program. The application should perform error-related actions and/or recover from the error. The status codes are of integer type and have the following format:

VSL_ERROR_<ERROR_NAME> - indicates VS errors common for all VS domains.

VSL_RNG_ERROR_<ERROR_NAME> - indicates VS RNG errors.

VS RNG errors are of negative values while warnings are of positive values. The status code of zero value indicates successful completion of the operation: VSL_ERROR_OK (or synonymic VSL_STATUS_OK).

Status Codes

Status Code	Description
Common VSL	
VSL_ERROR_OK, VSL_STATUS_OK	No error, execution is successful.
VSL_ERROR_BADARGS	Input argument value is not valid.
VSL_ERROR_CPU_NOT_SUPPORTED	CPU version is not supported.
VSL_ERROR_FEATURE_NOT_IMPLEMENTED	Feature invoked is not implemented.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory.
VSL_ERROR_NULL_PTR	Input pointer argument is NULL.
VSL_ERROR_UNKNOWN	Unknown error.
VS RNG Specific	
VSL_RNG_ERROR_BAD_FILE_FORMAT	File format is unknown.
VSL_RNG_ERROR_BAD_MEM_FORMAT	Descriptive random stream format is unknown.
VSL_RNG_ERROR_BAD_NBITS	The value in NBits field is bad.
VSL_RNG_ERROR_BAD_NSEEDS	The value in NSeeds field is bad.

Status Code	Description
VSL_RNG_ERROR_BAD_STREAM	The random stream is invalid.
VSL_RNG_ERROR_BAD_STREAM_STATE_SIZE	The value in <code>StreamStateSize</code> field is bad.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.
VSL_RNG_ERROR_BAD_WORD_SIZE	The value in <code>WordSize</code> field is bad.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_BRNG_TABLE_FULL	Registration cannot be completed due to lack of free entries in the table of registered BRNGs.
VSL_RNG_ERROR_BRNGS_INCOMPATIBLE	Two BRNGs are not compatible for the operation.
VSL_RNG_ERROR_FILE_CLOSE	Error in closing the file.
VSL_RNG_ERROR_FILE_OPEN	Error in opening the file.
VSL_RNG_ERROR_FILE_READ	Error in reading the file.
VSL_RNG_ERROR_FILE_WRITE	Error in writing the file.
VSL_RNG_ERROR_INVALID_ABSTRACT_STREAM	The abstract random stream is invalid.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is not valid.
VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns zero as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator is exceeded.
VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support Skip-Ahead method.
VSL_RNG_ERROR_SKIPAHEAD_EX_UNSUPPORTED	BRNG does not support advanced Skip-Ahead method.
VSL_RNG_ERROR_UNSUPPORTED_FILE_VER	File format version is not supported.
VSL_RNG_ERROR_NONDETERM_NOT_SUPPORTED	Non-deterministic random number generator is not supported on the CPU running the application.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number using non-deterministic random number generator exceeds threshold (see Section 7.2.1.12 <i>Non-deterministic</i> in [VS Notes] for more details)
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

Product and Performance Information

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

Notice revision #20201201

VS RNG Usage ModelIntel® oneMKL RNG Usage Model

A typical algorithm for VSoneMKL random number generators is as follows:

1. Create and initialize stream/streams. Functions `vslNewStream`, `vslNewStreamEx`, `vslCopyStream`, `vslCopyStreamState`, `vslLeapfrogStream`, `vslSkipAheadStream`, `vslSkipAheadStreamEx`.
2. Call one or more RNGs.
3. Process the output.
4. Delete the stream or streams with the function `vslDeleteStream`.

NOTE

You may reiterate steps 2-3. Random number streams may be generated for different threads.

The following example demonstrates generation of a random stream that is output of basic generator MT19937. The seed is equal to 777. The stream is used to generate 10,000 normally distributed random numbers in blocks of 1,000 random numbers with parameters $a = 5$ and $\sigma = 2$. Delete the streams after completing the generation. The purpose of the example is to calculate the sample mean for normal distribution with the given parameters.

Example of VS RNG Usage

```
#include <stdio.h>
#include "mkl_vsl.h"

int main()
{
    double r[1000]; /* buffer for random numbers */
    double s; /* average */
    VSLStreamStatePtr stream;
    int i, j;

    /* Initializing */
    s = 0.0;
    vslNewStream( &stream, VSL_BRNG_MT19937, 777 );

    /* Generating */
    for ( i=0; i<10; i++ ) {
        vdRngGaussian( VSL_RNG_METHOD_GAUSSIAN_ICDF, stream, 1000, r, 5.0, 2.0 );
        for ( j=0; j<1000; j++ ) {
            s += r[j];
        }
    }
    s /= 10000.0;

    /* Deleting the stream */
    vslDeleteStream( &stream );

    /* Printing results */
    printf( "Sample mean of normal distribution = %f\n", s );

    return 0;
}
```

Additionally, examples that demonstrate usage of VS random number generators are available in:

`${MKL}/examples/vslc/source`

Service Routines

Stream handling comprises routines for creating, deleting, or copying the streams and getting the index of a basic generator. A random stream can also be saved to and then read from a binary file. [Table "Service Routines"](#) lists all available service routines

Service Routines

Routine	Short Description
<code>vslNewStream</code>	Creates and initializes a random stream.
<code>vslNewStreamEx</code>	Creates and initializes a random stream for the generators with multiple initial conditions.
<code>vslNewAbstractStream</code>	Creates and initializes an abstract random stream for integer arrays.
<code>vslNewAbstractStream</code>	Creates and initializes an abstract random stream for double precision floating-point arrays.
<code>vslNewAbstractStream</code>	Creates and initializes an abstract random stream for single precision floating-point arrays.
<code>vslDeleteStream</code>	Deletes previously created stream.
<code>vslCopyStream</code>	Copies a stream to another stream.
<code>vslCopyStreamState</code>	Creates a copy of a random stream state.
<code>vslSaveStreamF</code>	Writes a stream to a binary file.
<code>vslLoadStreamF</code>	Reads a stream from a binary file.
<code>vslSaveStreamM</code>	Writes a random stream descriptive data, including state, to a memory buffer.
<code>vslLoadStreamM</code>	Creates a new stream and reads stream descriptive data, including state, from the memory buffer.
<code>vslGetStreamSize</code>	Computes size of memory necessary to hold the random stream.
<code>vslLeapfrogStream</code>	Initializes the stream by the leapfrog method to generate a subsequence of the original sequence.
<code>vslSkipAheadStream</code>	Initializes the stream by the skip-ahead method.
<code>vslSkipAheadStreamEx</code>	Initializes the stream by the advanced skip-ahead method.
<code>vslGetStreamStateBrng</code>	Obtains the index of the basic generator responsible for the generation of a given random stream.
<code>vslGetNumRegBrngs</code>	Obtains the number of currently registered basic generators.

Most of the generator-based work comprises three basic steps:

1. Creating and initializing a stream (`vslNewStream`, `vslNewStreamEx`, `vslCopyStream`, `vslCopyStreamState`, `vslLeapfrogStream`, `vslSkipAheadStream`, `vslSkipAheadStreamEx`).
2. Generating random numbers with given distribution, see [Distribution Generators](#).
3. Deleting the stream (`vslDeleteStream`).

Note that you can concurrently create multiple streams and obtain random data from one or several generators by using the stream state. You must use the [vslDeleteStream](#) function to delete all the streams afterwards.

vslNewStream

Creates and initializes a random stream.

Syntax

```
status = vslNewStream( &stream, brng, seed );
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>brng</i>	<code>const MKL_INT</code>	Index of the basic generator to initialize the stream. See Table Values of <i>brng</i> parameter for specific value.
<i>seed</i>	<code>const MKL_UINT</code>	Initial condition of the stream. In the case of a quasi-random number generator seed parameter is used to set the dimension. If the dimension is greater than the dimension that <i>brng</i> can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
<i>stream</i>	<code>VSLStreamStatePtr*</code>	Stream state descriptor

Description

For a basic generator with number *brng*, this function creates a new stream and initializes it with a 32-bit seed. The seed is an initial value used to select a particular sequence generated by the basic generator *brng*. The function is also applicable for generators with multiple initial conditions. Use this function to create and initialize a new stream with a 32-bit seed only. If you need to provide multiple initial conditions such as several 32-bit or wider seeds, use the function [vslNewStreamEx](#). See [VS Notes](#) for a more detailed description of stream initialization for different basic generators.

NOTE

This function is not applicable for abstract basic random number generators. Please use [vslNewAbstractStream](#), [vslsNewAbstractStream](#) or [vsldNewAbstractStream](#) to utilize integer, single-precision or double-precision external random data respectively.

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Non-deterministic random number generator is not supported.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vslNewStreamEx

Creates and initializes a random stream for generators with multiple initial conditions.

Syntax

```
status = vslNewStreamEx( &stream, brng, n, params );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>brng</i>	const MKL_INT	Index of the basic generator to initialize the stream. See Table "Values of <i>brng</i> parameter" for specific value.
<i>n</i>	const MKL_INT	Number of initial conditions contained in <i>params</i>
<i>params</i>	const unsigned int	Array of initial conditions necessary for the basic generator <i>brng</i> to initialize the stream. In the case of a quasi-random number generator only the first element in <i>params</i> parameter is used to set the dimension. If the dimension is greater than the dimension that <i>brng</i> can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
<i>stream</i>	VSLStreamStatePtr*	Stream state descriptor

Description

The `vslNewStreamEx` function provides an advanced tool to set the initial conditions for a basic generator if its input arguments imply several initialization parameters. Initial values are used to select a particular sequence generated by the basic generator *brng*. Whenever possible, use `vslNewStream`, which is analogous to `vslNewStreamEx` except that it takes only one 32-bit initial condition. In particular, `vslNewStreamEx` may be used to initialize the state table in Generalized Feedback Shift Register Generators (GFSRs). A more detailed description of this issue can be found in [VS Notes](#).

This function is also used to pass user-defined initialization parameters of quasi-random number generators into the library. See [VS Notes](#) for the format for their passing and registration in VS.

NOTE

This function is not applicable for abstract basic random number generators. Please use [vsliNewAbstractStream](#), [vslsNewAbstractStream](#) or [vsldNewAbstractStream](#) to utilize integer, single-precision or double-precision external random data respectively.

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Non-deterministic random number generator is not supported.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vsliNewAbstractStream

Creates and initializes an abstract random stream for integer arrays.

Syntax

```
status = vsliNewAbstractStream( &stream, n, ibuf, icallback );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Size of the array <i>ibuf</i>
<i>ibuf</i>	const unsigned int	Array of <i>n</i> 32-bit integers
<i>icallback</i>		Pointer to the callback function used for <i>ibuf</i> update

NOTE

Format of the callback function:

```
int iUpdateFunc( VSLStreamStatePtrstream, int* n, unsigned int ibuf[], int* nmin, int* nmax, int* idx );
```

The callback function returns the number of elements in the array actually updated by the function. [Table icallback Callback Function Parameters](#) gives the description of the callback function parameters.

icallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract random stream descriptor
<i>n</i>	Size of <i>ibuf</i>
<i>ibuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>ibuf</i> to start update $0 \leq idx < n$.

Output Parameters

Name	Type	Description
<i>stream</i>	VSLStreamStatePtr*	Descriptor of the stream state structure

Description

The `vsliNewAbstractStream` function creates a new abstract stream and associates it with an integer array *ibuf* and your callback function *icallback* that is intended for updating of *ibuf* content.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BADARGS	Parameter <i>n</i> is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

vsldNewAbstractStream

Creates and initializes an abstract random stream for double precision floating-point arrays.

Syntax

```
status = vsldNewAbstractStream( &stream, n, dbuf, a, b, dcallback );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Size of the array <i>dbuf</i>
<i>dbuf</i>	const double	Array of <i>n</i> double precision floating-point random numbers with uniform distribution over interval (<i>a</i> , <i>b</i>)

Name	Type	Description
<i>a</i>	const double	Left boundary <i>a</i>
<i>b</i>	const double	Right boundary <i>b</i>
<i>dcallback</i>	See <i>Note</i> below	Pointer to the callback function used for update of the array <i>dbuf</i>

Output Parameters

Name	Type	Description
<i>stream</i>	VSLStreamStatePtr*	Descriptor of the stream state structure

NOTE

Format of the callback function:

```
int dUpdateFunc( VSLStreamStatePtr stream, int* n, double dbuf[], int* nmin, int* nmax,
int* idx );
```

The callback function returns the number of elements in the array actually updated by the function. [Table dcallback Callback Function Parameters](#) gives the description of the callback function parameters.

dcallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract random stream descriptor
<i>n</i>	Size of <i>dbuf</i>
<i>dbuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>dbuf</i> to start update $0 \leq idx < n$.

Description

The `vsldNewAbstractStream` function creates a new abstract stream for double precision floating-point arrays with random numbers of the uniform distribution over interval (*a*,*b*). The function associates the stream with a double precision array *dbuf* and your callback function *dcallback* that is intended for updating of *dbuf* content.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BADARGS	Parameter <i>n</i> is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

vslsNewAbstractStream

Creates and initializes an abstract random stream for single precision floating-point arrays.

Syntax

```
status = vslsNewAbstractStream( &stream, n, sbuf, a, b, scallback );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT	Size of the array <i>sbuf</i>
<i>sbuf</i>	const float	Array of <i>n</i> single precision floating-point random numbers with uniform distribution over interval (<i>a</i> , <i>b</i>)
<i>a</i>	const float	Left boundary <i>a</i>
<i>b</i>	const float	Right boundary <i>b</i>
<i>scallback</i>	See <i>Note</i> below	Pointer to the callback function used for update of the array <i>sbuf</i>

Output Parameters

Name	Type	Description
<i>stream</i>	VSLStreamStatePtr*	Descriptor of the stream state structure

NOTE

Format of the callback function in C:

```
int sUpdateFunc( VSLStreamStatePtr stream, int* n, float sbuf[], int* nmin, int* nmax,
int* idx );
```

The callback function returns the number of elements in the array actually updated by the function. [Table *scallback* Callback Function Parameters](#) gives the description of the callback function parameters.

scallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract random stream descriptor
<i>n</i>	Size of <i>sbuf</i>
<i>sbuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>sbuf</i> to start update $0 \leq idx < n$.

Description

The `vslsNewAbstractStream` function creates a new abstract stream for single precision floating-point arrays with random numbers of the uniform distribution over interval (a,b) . The function associates the stream with a single precision array `sbuf` and your callback function `scallback` that is intended for updating of `sbuf` content.

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_BADARGS</code>	Parameter <i>n</i> is not positive.
<code>VSL_ERROR_MEM_FAILURE</code>	System cannot allocate memory for <i>stream</i> .
<code>VSL_ERROR_NULL_PTR</code>	Either buffer or callback function parameter is a <code>NULL</code> pointer.

vslDeleteStream

Deletes a random stream.

Syntax

```
status = vslDeleteStream( &stream );
```

Include Files

- `mkl.h`

Input/Output Parameters

Name	Type	Description
<i>stream</i>	<code>VSLStreamStatePtr*</code>	Stream state descriptor. Must have non-zero value. After the stream is successfully deleted, the pointer is set to <code>NULL</code> .

Description

The function deletes the random stream created by one of the initialization functions.

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<i>stream</i> parameter is a <code>NULL</code> pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is not a valid random stream.

vslCopyStream

Creates a copy of a random stream.

Syntax

```
status = vslCopyStream( &newstream, srcstream );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>srcstream</i>	<code>const VSLStreamStatePtr</code>	Pointer to the stream state structure to be copied

Output Parameters

Name	Type	Description
<i>newstream</i>	<code>VSLStreamStatePtr*</code>	Copied random stream descriptor

Description

The function creates an exact copy of *srcstream* and stores its descriptor to *newstream*.

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<i>srcstream</i> parameter is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>srcstream</i> is not a valid random stream.
<code>VSL_ERROR_MEM_FAILURE</code>	System cannot allocate memory for <i>newstream</i> .

`vslCopyStreamState`

Creates a copy of a random stream state.

Syntax

```
status = vslCopyStreamState( deststream, srcstream );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>srcstream</i>	<code>const VSLStreamStatePtr</code>	Pointer to the stream state structure, from which the state structure is copied

Output Parameters

Name	Type	Description
<i>deststream</i>	<code>VSLStreamStatePtr</code>	Pointer to the stream state structure where the stream state is copied

Description

The `vslCopyStreamState` function copies a stream state from *srcstream* to the existing *deststream* stream. Both the streams should be generated by the same basic generator. An error message is generated when the index of the BRNG that produced *deststream* stream differs from the index of the BRNG that generated *srcstream* stream.

Unlike `vslCopyStream` function, which creates a new stream and copies both the stream state and other data from *srcstream*, the function `vslCopyStreamState` copies only *srcstream* stream state data to the generated *deststream* stream.

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	Either <i>srcstream</i> or <i>deststream</i> is a <code>NULL</code> pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	Either <i>srcstream</i> or <i>deststream</i> is not a valid random stream.
<code>VSL_RNG_ERROR_BRNGS_INCOMPATIBLE</code>	BRNG associated with <i>srcstream</i> is not compatible with BRNG associated with <i>deststream</i> .

vslSaveStreamF

Writes random stream descriptive data, including stream state, to binary file.

Syntax

```
errstatus = vslSaveStreamF( stream, fname );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>stream</i>	<code>const VSLStreamStatePtr</code>	Random stream to be written to the file
<i>fname</i>	<code>const char*</code>	File name specified as a null-terminated string

Output Parameters

Name	Type	Description
<i>errstatus</i>	<code>int</code>	Error status of the operation

Description

The `vslSaveStreamF` function writes the random stream descriptive data, including the stream state, to the binary file. Random stream descriptive data is saved to the binary file with the name *fname*. The random stream *stream* must be a valid stream created by `vslNewStream`-like or `vslCopyStream`-like service routines. If the stream cannot be saved to the file, *errstatus* has a non-zero value. The random stream can be read from the binary file using the `vslLoadStreamF` function.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either <i>fname</i> or <i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_RNG_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_RNG_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.

vslLoadStreamF

Creates new stream and reads stream descriptive data, including stream state, from binary file.

Syntax

```
errstatus = vslLoadStreamF( &stream, fname );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>fname</i>	const char*	File name specified as a null-terminated string

Output Parameters

Name	Type	Description
<i>stream</i>	VSLStreamStatePtr*	Pointer to a new random stream
<i>errstatus</i>	int	Error status of the operation

Description

The `vslLoadStreamF` function creates a new stream and reads stream descriptive data, including the stream state, from the binary file. A new random stream is created using the stream descriptive data from the binary file with the name *fname*. If the stream cannot be read (for example, an I/O error occurs or the file format is invalid), *errstatus* has a non-zero value. To save random stream to the file, use `vslSaveStreamF` function.

Caution

Calling `vslLoadStreamF` with a previously initialized *stream* pointer can have unintended consequences such as a memory leak. To initialize a stream which has been in use until calling `vslLoadStreamF`, you should call the `vslDeleteStream` function first to deallocate the resources.

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>fname</i> is a NULL pointer.
VSL_RNG_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_RNG_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_RNG_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.
VSL_RNG_ERROR_BAD_FILE_FORMAT	Unknown file format.
VSL_RNG_ERROR_UNSUPPORTED_FILE_VER	File format version is unsupported.
VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Non-deterministic random number generator is not supported.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vslSaveStreamM

Writes random stream descriptive data, including stream state, to a memory buffer.

Syntax

```
errstatus = vslSaveStreamM( stream, memptr );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>stream</i>	const VSLStreamStatePtr	Random stream to be written to the memory
<i>memptr</i>	char*	Memory buffer to save random stream descriptive data to

Output Parameters

Name	Type	Description
<i>errstatus</i>	int	Error status of the operation

Description

The `vslSaveStreamM` function writes the random stream descriptive data, including the stream state, to the memory at `memptr`. Random stream `stream` must be a valid stream created by `vslNewStream`-like or `vslCopyStream`-like service routines. The `memptr` parameter must be a valid pointer to the memory of size sufficient to hold the random stream `stream`. Use the service routine `vslGetStreamSize` to determine this amount of memory.

If the stream cannot be saved to the memory, `errstatus` has a non-zero value. The random stream can be read from the memory pointed by `memptr` using the `vslLoadStreamM` function.

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	Either <code>memptr</code> or <code>stream</code> is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<code>stream</code> is a NULL pointer.

vslLoadStreamM

Creates a new stream and reads stream descriptive data, including stream state, from the memory buffer.

Syntax

```
errstatus = vslLoadStreamM( &stream, memptr );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>memptr</code>	<code>const char*</code>	Memory buffer to load random stream descriptive data from

Output Parameters

Name	Type	Description
<code>stream</code>	<code>VSLStreamStatePtr*</code>	Pointer to a new random stream
<code>errstatus</code>	<code>int</code>	Error status of the operation

Description

The `vslLoadStreamM` function creates a new stream and reads stream descriptive data, including the stream state, from the memory buffer. A new random stream is created using the stream descriptive data from the memory pointer by `memptr`. If the stream cannot be read (for example, `memptr` is invalid), `errstatus` has a non-zero value. To save random stream to the memory, use `vslSaveStreamM` function. Use the service routine `vslGetStreamSize` to determine the amount of memory sufficient to hold the random stream.

Caution

Calling `LoadStreamM` with a previously initialized `stream` pointer can have unintended consequences such as a memory leak. To initialize a stream which has been in use until calling `vslLoadStreamM`, you should call the `vslDeleteStream` function first to deallocate the resources.

Product and Performance Information

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

Notice revision #20201201

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<code>memptr</code> is a <code>NULL</code> pointer.
<code>VSL_ERROR_MEM_FAILURE</code>	System cannot allocate memory for internal needs.
<code>VSL_RNG_ERROR_BAD_MEM_FORMAT</code>	Descriptive random stream format is unknown.
<code>VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED</code>	Non-deterministic random number generator is not supported.
<code>VSL_RNG_ERROR_ARS5_NOT_SUPPORTED</code>	ARS-5 random number generator is not supported on the CPU running the application.

vslGetStreamSize

Computes size of memory necessary to hold the random stream.

Syntax

```
memsize = vslGetStreamSize( stream );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>stream</code>	<code>const VSLStreamStatePtr</code>	Random stream

Output Parameters

Name	Type	Description
<code>memsize</code>	<code>int</code>	Amount of memory in bytes necessary to hold descriptive data of random stream <code>stream</code>

Description

The `vslGetStreamSize` function returns the size of memory in bytes which is necessary to hold the given random stream. Use the output of the function to allocate the buffer to which you will save the random stream by means of the `vslSaveStreamM` function.

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is a NULL pointer.

`vslLeapfrogStream`

Initializes a stream using the leapfrog method.

Syntax

```
status = vslLeapfrogStream( stream, k, nstreams );
```

Include Files

- `mkl.h`

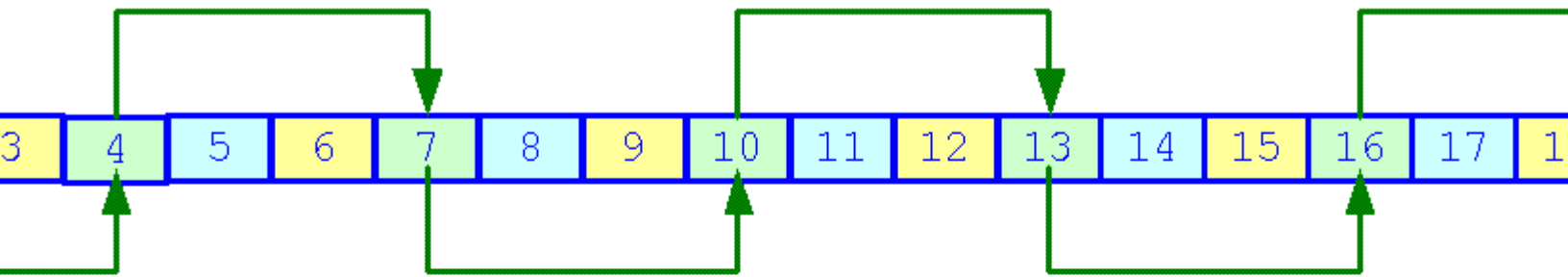
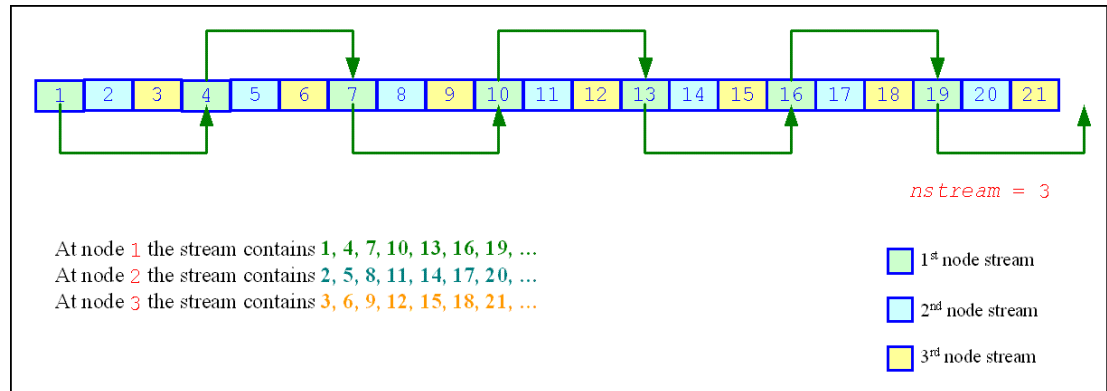
Input Parameters

Name	Type	Description
<i>stream</i>	<code>VSLStreamStatePtr</code>	Pointer to the stream state structure to which leapfrog method is applied
<i>k</i>	<code>const MKL_INT</code>	Index of the computational node, or stream number
<i>nstreams</i>	<code>const MKL_INT</code>	Largest number of computational nodes, or stride

Description

The `vslLeapfrogStream` function generates random numbers in a random stream with non-unit stride. This feature is particularly useful in distributing random numbers from the original stream across the *nstreams* buffers without generating the original random sequence with subsequent manual distribution.

One of the important applications of the leapfrog method is splitting the original sequence into non-overlapping subsequences across *nstreams* computational nodes. The function initializes the original random stream (see [Figure "Leapfrog Method"](#)) to generate random numbers for the computational node *k*, $0 \leq k < nstreams$, where *nstreams* is the largest number of computational nodes used.

`__border__top`**Leapfrog Method**

the stream contains 1, 4, 7, 10, 13, 16, 19, ...
 the stream contains 2, 5, 8, 11, 14, 17, 20, ...
 the stream contains 3, 6, 9, 12, 15, 18, 21, ...

The leapfrog method is supported only for those basic generators that allow splitting elements by the leapfrog method, which is more efficient than simply generating them by a generator with subsequent manual distribution across computational nodes. See [VS Notes](#) for details.

For quasi-random basic generators, the leapfrog method allows generating individual components of quasi-random vectors instead of whole quasi-random vectors. In this case *nstreams* parameter should be equal to the dimension of the quasi-random vector while *k* parameter should be the index of a component to be generated ($0 \leq k < nstreams$). Other parameters values are not allowed.

The following code illustrates the initialization of three independent streams using the leapfrog method:

Code for Leapfrog Method

```
...
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;

/* Creating 3 identical streams */
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);
status = vslCopyStream(&stream2, stream1);
status = vslCopyStream(&stream3, stream1);

/* Leapfrogging the streams
*/
status = vslLeapfrogStream(stream1, 0, 3);
status = vslLeapfrogStream(stream2, 1, 3);
status = vslLeapfrogStream(stream3, 2, 3);

/* Generating random numbers
*/
...
/* Deleting the streams
*/
status = vslDeleteStream(&stream1);
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...
```

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.

vslSkipAheadStream

Initializes a stream using the block-splitting method.

Syntax

```
status = vslSkipAheadStream( stream, nskip);
```

Include Files

- mkl.h

Input Parameters

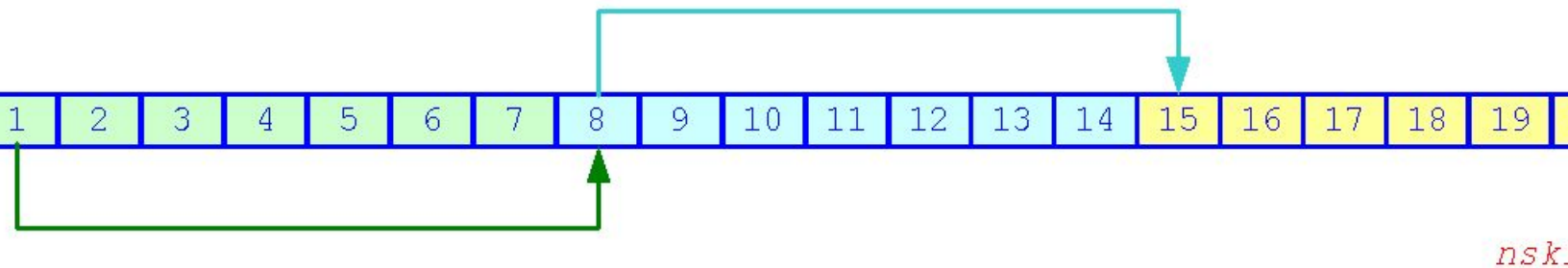
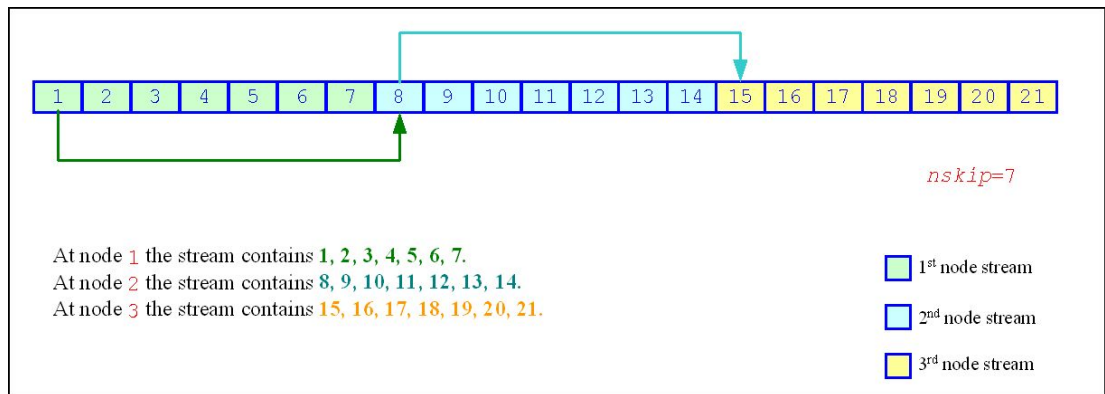
Name	Type	Description
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure to which block-splitting method is applied
<i>nskip</i>	const long long int	Number of skipped elements

Description

The `vslSkipAheadStream` function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is n_{skip} , then the original random sequence may be split by `vslSkipAheadStream` into non-overlapping blocks of n_{skip} size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method. (see [Figure "Block-Splitting Method"](#)).

__border__top

Block-Splitting Method



At node 1 the stream contains 1, 2, 3, 4, 5, 6, 7.
 At node 2 the stream contains 8, 9, 10, 11, 12, 13, 14.
 At node 3 the stream contains 15, 16, 17, 18, 19, 20, 21.

- 1st node stream
- 2nd node stream
- 3rd node stream

The skip-ahead method is supported only for those basic generators that allow skipping elements by the skip-ahead method, which is more efficient than simply generating them by generator with subsequent manual skipping. See [VS Notes](#) for details.

Please note that for quasi-random basic generators the skip-ahead method works with components of quasi-random vectors rather than with whole quasi-random vectors. Therefore, to skip `NS` quasi-random vectors, set the `nskip` parameter equal to the `NS*DIMEN`, where `DIMEN` is the dimension of the quasi-random vector. If this operation results in exceeding the period of the quasi-random number generator, which is $2^{32}-1$, the library returns the `VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED` error code.

The following code illustrates how to initialize three independent streams using the `vslSkipAheadStream` function:

Code for Block-Splitting Method

```
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;

/* Creating the 1st stream
*/
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);

/* Skipping ahead by 7 elements the 2nd stream */
status = vslCopyStream(&stream2, stream1);
status = vslSkipAheadStream(stream2, 7);

/* Skipping ahead by 7 elements the 3rd stream */
status = vslCopyStream(&stream3, stream2);
status = vslSkipAheadStream(stream3, 7);

/* Generating random numbers
*/
...
/* Deleting the streams
*/
status = vslDeleteStream(&stream1);
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...
```

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<code>stream</code> is a <code>NULL</code> pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<code>stream</code> is not a valid random stream.
<code>VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED</code>	BRNG does not support the Skip-Ahead method.
<code>VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED</code>	Period of the quasi-random number generator is exceeded.

vslSkipAheadStreamEx

Initializes a stream using the block-splitting method with partitioned number of skipped elements.

Syntax

```
status = vslSkipAheadStreamEx( stream, n, nskip);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure to which block-splitting method is applied
<i>n</i>	const MKL_INT	Number of summands in <i>nskip</i>
<i>nskip</i>	const MKL_UINT64[]	Partitioned number of skipped elements

Description

The `vslSkipAheadStreamEx` function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is *nskip*, then the original random sequence may be split by `vslSkipAheadStreamEx` into non-overlapping blocks of *nskip* size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method.

Use this function when the number of elements to skip in a random stream is greater than 2^{63} . Prior calls to the function represent the number of skipped elements with array of size *n* as shown below:

$$nskip[0] + nskip[1] * 2^{64} + nskip[2] * 2^{128} + \dots + nskip[n-1] * 2^{(64 * (n-1))};$$

When the number of skipped elements is less than 2^{63} you can use either `vslSkipAheadStreamEx` or `vslSkipAheadStream`. The following code illustrates how to initialize three independent streams using the `vslSkipAheadStreamEx` function:

```
VSLStreamStatePtr stream1; VSLStreamStatePtr stream2; VSLStreamStatePtr stream3;

/* Creating the 1st stream
*/
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);

/* To skip 2^64 elements in the random stream SkipAheadStreamEx(nskip) function should be called
with nskip represented as nskip = 2^64 = 0 + 1 * 2^64
*/
MKL_UINT64 nskip[2];
nskip[0]=0;
nskip[1]=1;

/* Skipping ahead by 2^64 elements the 2nd stream
*/
status = vslCopyStream(&stream2, stream1);
status = vslSkipAheadStreamEx (stream2, 2, nskip);

/* Skipping ahead by 2^64 elements the 3rd stream
*/
status = vslCopyStream(&stream3, stream2);
status = vslSkipAheadStreamEx (stream3, 2, nskip);

/* Generating random numbers
*/
...
/* Deleting the streams
```

```
*/
status = vslDeleteStream(&stream1);
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...
```

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support the advanced Skip-Ahead method.

vslGetStreamStateBrng

Returns index of a basic generator used for generation of a given random stream.

Syntax

```
brng = vslGetStreamStateBrng( stream );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>stream</i>	const VSLStreamStatePtr	Pointer to the stream state structure

Output Parameters

Name	Type	Description
<i>brng</i>	int	Index of the basic generator assigned for the generation of <i>stream</i> ; negative in case of an error

Description

The `vslGetStreamStateBrng` function retrieves the index of a basic generator used for generation of a given random stream.

Return Values

VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

vslGetNumRegBrngs

Obtains the number of currently registered basic generators.

Syntax

```
nregbrngs = vslGetNumRegBrngs( void );
```

Include Files

- `mkl.h`

Output Parameters

Name	Type	Description
<code>nregbrngs</code>	<code>int</code>	Number of basic generators registered at the moment of the function call

Description

The `vslGetNumRegBrngs` function obtains the number of currently registered basic generators. Whenever user registers a user-designed basic generator, the number of registered basic generators is incremented. The maximum number of basic generators that can be registered is determined by the `VSL_MAX_REG_BRNGS` parameter.

Distribution Generators

oneMKLVS routines are used to generate random numbers with different types of distribution. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence and the explanation of input and output parameters. [Table "Continuous Distribution Generators"](#) and [Table "Discrete Distribution Generators"](#) list the random number generator routines with data types and output distributions, and sets correspondence between data types of the generator routines and the basic random number generators.

Continuous Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
<code>vRngUniform</code>	<code>s, d</code>	<code>s, d</code>	Uniform continuous distribution on the interval $[a,b)$
<code>vRngGaussian</code>	<code>s, d</code>	<code>s, d</code>	Normal (Gaussian) distribution
<code>vRngGaussianMV</code>	<code>s, d</code>	<code>s, d</code>	Normal (Gaussian) multivariate distribution
<code>vRngExponential</code>	<code>s, d</code>	<code>s, d</code>	Exponential distribution
<code>vRngLaplace</code>	<code>s, d</code>	<code>s, d</code>	Laplace distribution (double exponential distribution)
<code>vRngWeibull</code>	<code>s, d</code>	<code>s, d</code>	Weibull distribution
<code>vRngCauchy</code>	<code>s, d</code>	<code>s, d</code>	Cauchy distribution
<code>vRngRayleigh</code>	<code>s, d</code>	<code>s, d</code>	Rayleigh distribution
<code>vRngLognormal</code>	<code>s, d</code>	<code>s, d</code>	Lognormal distribution
<code>vRngGumbel</code>	<code>s, d</code>	<code>s, d</code>	Gumbel (extreme value) distribution
<code>vRngGamma</code>	<code>s, d</code>	<code>s, d</code>	Gamma distribution
<code>vRngBeta</code>	<code>s, d</code>	<code>s, d</code>	Beta distribution

Type of Distribution	Data Types	BRNG Data Type	Description
<code>vRngChiSquare</code>	<code>s</code> , <code>d</code>	<code>s</code> , <code>d</code>	Chi-Square distribution

Discrete Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
<code>vRngUniform</code>	<code>i</code>	<code>d</code>	Uniform discrete distribution on the interval $[a,b)$
<code>vRngUniformBits</code>	<code>i</code>	<code>i</code>	Underlying BRNG integer recurrence
<code>vRngUniformBits32</code>	<code>i</code>	<code>i</code>	Uniformly distributed bits in 32-bit chunks
<code>vRngUniformBits64</code>	<code>i</code>	<code>i</code>	Uniformly distributed bits in 64-bit chunks
<code>vRngBernoulli</code>	<code>i</code>	<code>s</code>	Bernoulli distribution
<code>vRngGeometric</code>	<code>i</code>	<code>s</code>	Geometric distribution
<code>vRngBinomial</code>	<code>i</code>	<code>d</code>	Binomial distribution
<code>vRngHypergeometric</code>	<code>i</code>	<code>d</code>	Hypergeometric distribution
<code>vRngPoisson</code>	<code>i</code>	<code>s</code> (for VSL_RNG_METHOD_POISSON_POISNORM) <code>s</code> (for distribution parameter $\lambda \geq 27$) and <code>d</code> (for $\lambda < 27$) (for VSL_RNG_METHOD_POISSON_PTPE)	Poisson distribution
<code>vRngPoisson</code>	<code>i</code>	<code>s</code>	Poisson distribution with varying mean
<code>vRngNegBinomial</code>	<code>i</code>	<code>d</code>	Negative binomial distribution, or Pascal distribution
<code>vRngMultinomial</code>	<code>i</code>	<code>d</code>	Multinomial distribution

Modes of random number generation

The library provides two modes of random number generation, accurate and fast. Accurate generation mode is intended for the applications that are highly demanding to accuracy of calculations. When used in this mode, the generators produce random numbers lying completely within definitional domain for all values of the distribution parameters. For example, random numbers obtained from the generator of continuous distribution that is uniform on interval $[a,b]$ belong to this interval irrespective of what a and b values may be. Fast mode provides high performance of generation and also guarantees that generated random numbers

belong to the definitional domain except for some specific values of distribution parameters. The generation mode is set by specifying relevant value of the method parameter in generator routines. List of distributions that support accurate mode of generation is given in the table below.

Distribution Generators Supporting Accurate Mode

Type of Distribution	Data Types
<code>vRngUniform</code>	s, d
<code>vRngExponential</code>	s, d
<code>vRngWeibull</code>	s, d
<code>vRngRayleigh</code>	s, d
<code>vRngLognormal</code>	s, d
<code>vRngGamma</code>	s, d
<code>vRngBeta</code>	s, d

See additional details about accurate and fast mode of random number generation in [VS Notes](#).

New method names

The current version of oneMKL has a modified structure of VS RNG method names. (See [RNG Naming Conventions](#) for details.) The old names are kept for backward compatibility. The tables below set correspondence between the new and legacy method names for VS random number generators.

Method Names for Continuous Distribution Generators

RNG	Legacy Method Name	New Method Name
<code>vRngUniform</code>	VSL_METHOD_SUNIFORM_STD, VSL_METHOD_DUNIFORM_STD, VSL_METHOD_SUNIFORM_STD_ACCURATE, VSL_METHOD_DUNIFORM_STD_ACCURATE	VSL_RNG_METHOD_UNIFORM_STD, VSL_RNG_METHOD_UNIFORM_STD_ACCURATE
<code>vRngGaussian</code>	VSL_METHOD_SGAUSSIAN_BOXMULLER, VSL_METHOD_SGAUSSIAN_BOXMULLER2, VSL_METHOD_SGAUSSIAN_ICDF, VSL_METHOD_DGAUSSIAN_BOXMULLER, VSL_METHOD_DGAUSSIAN_BOXMULLER2, VSL_METHOD_DGAUSSIAN_ICDF	VSL_RNG_METHOD_GAUSSIAN_BOXMULLER, VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2, VSL_RNG_METHOD_GAUSSIAN_ICDF
<code>vRngGaussianMV</code>	VSL_METHOD_SGAUSSIANMV_BOXMULLER, VSL_METHOD_SGAUSSIANMV_BOXMULLER2, VSL_METHOD_SGAUSSIANMV_ICDF, VSL_METHOD_DGAUSSIANMV_BOXMULLER, VSL_METHOD_DGAUSSIANMV_BOXMULLER2, VSL_METHOD_DGAUSSIANMV_ICDF	VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER, VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2, VSL_RNG_METHOD_GAUSSIANMV_ICDF
<code>vRngExponential</code>	VSL_METHOD_SEXPONENTIAL_ICDF, VSL_METHOD_DEXPONENTIAL_ICDF, VSL_METHOD_SEXPONENTIAL_ICDF_ACCURATE, VSL_METHOD_DEXPONENTIAL_ICDF_ACCURATE	VSL_RNG_METHOD_EXPONENTIAL_ICDF, VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE

RNG	Legacy Method Name	New Method Name
<code>vRngLaplace</code>	VSL_METHOD_SLAPLACE_ICDF, VSL_METHOD_DLAPLACE_ICDF	VSL_RNG_METHOD_LAPLACE_ICDF
<code>vRngWeibull</code>	VSL_METHOD_SWEIBULL_ICDF, VSL_METHOD_DWEIBULL_ICDF, VSL_METHOD_SWEIBULL_ICDF_ACCURATE, VSL_METHOD_DWEIBULL_ICDF_ACCURATE	VSL_RNG_METHOD_WEIBULL_ICDF, VSL_RNG_METHOD_WEIBULL_ICDF_ACCURATE
<code>vRngCauchy</code>	VSL_METHOD_SCAUCHY_ICDF, VSL_METHOD_DCAUCHY_ICDF	VSL_RNG_METHOD_CAUCHY_ICDF
<code>vRngRayleigh</code>	VSL_METHOD_SRAYLEIGH_ICDF, VSL_METHOD_DRAYLEIGH_ICDF, VSL_METHOD_SRAYLEIGH_ICDF_ACCURATE, VSL_METHOD_DRAYLEIGH_ICDF_ACCURATE	VSL_RNG_METHOD_RAYLEIGH_ICDF, VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE
<code>vRngLognormal</code>	VSL_METHOD_SLOGNORMAL_BOXMULLER2, VSL_METHOD_DLOGNORMAL_BOXMULLER2, VSL_METHOD_SLOGNORMAL_BOXMULLER2_ACCURATE, VSL_METHOD_DLOGNORMAL_BOXMULLER2_ACCURATE	VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2, VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE
	VSL_METHOD_SLOGNORMAL_ICDF, VSL_METHOD_DLOGNORMAL_ICDF, VSL_METHOD_SLOGNORMAL_ICDF_ACCURATE, VSL_METHOD_DLOGNORMAL_ICDF_ACCURATE	VSL_RNG_METHOD_LOGNORMAL_ICDF, VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCURATE
<code>vRngGumbel</code>	VSL_METHOD_SGUMBEL_ICDF, VSL_METHOD_DGUMBEL_ICDF	VSL_RNG_METHOD_GUMBEL_ICDF
<code>vRngGamma</code>	VSL_METHOD_SGAMMA_GNORM, VSL_METHOD_DGAMMA_GNORM, VSL_METHOD_SGAMMA_GNORM_ACCURATE, VSL_METHOD_DGAMMA_GNORM_ACCURATE	VSL_RNG_METHOD_GAMMA_GNORM, VSL_RNG_METHOD_GAMMA_GNORM_ACCURATE
<code>vRngBeta</code>	VSL_METHOD_SBETA_CJA, VSL_METHOD_DBETA_CJA, VSL_METHOD_SBETA_CJA_ACCURATE, VSL_METHOD_DBETA_CJA_ACCURATE	VSL_RNG_METHOD_BETA_CJA, VSL_RNG_METHOD_BETA_CJA_ACCURATE

Method Names for Discrete Distribution Generators

RNG	Legacy Method Name	New Method Name
<code>vRngUniform</code>	VSL_METHOD_IUNIFORM_STD	VSL_RNG_METHOD_UNIFORM_STD
<code>vRngUniformBits</code>	VSL_METHOD_IUNIFORMBITS_STD	VSL_RNG_METHOD_UNIFORMBITS_STD
<code>vRngBernoulli</code>	VSL_METHOD_IBERNOULLI_ICDF	VSL_RNG_METHOD_BERNOULLI_ICDF
<code>vRngGeometric</code>	VSL_METHOD_IGEOMETRIC_ICDF	VSL_RNG_METHOD_GEOMETRIC_ICDF
<code>vRngBinomial</code>	VSL_METHOD_IBINOMIAL_BTPE	VSL_RNG_METHOD_BINOMIAL_BTPE

RNG	Legacy Method Name	New Method Name
<code>vRngHypergeomet</code>	<code>VSL_METHOD_IHYPERGEOMETRIC_H2PE</code>	<code>VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE</code>
<code>vRngPoisson</code>	<code>VSL_METHOD_IPOISSON_PTPE,</code> <code>VSL_METHOD_IPOISSON_POISNORM</code>	<code>VSL_RNG_METHOD_POISSON_PTPE,</code> <code>VSL_RNG_METHOD_POISSON_POISNORM</code>
<code>vRngPoissonV</code>	<code>VSL_METHOD_IPOISSONV_POISNORM</code>	<code>VSL_RNG_METHOD_POISSONV_POISNORM</code>
<code>vRngNegBinomial</code>	<code>VSL_METHOD_INEGBINOMIAL_NBAR</code>	<code>VSL_RNG_METHOD_NEGBINOMIAL_NBAR</code>

Continuous Distributions

This section describes routines for generating random numbers with continuous distribution.

`vRngUniform` *Continuous Distribution Generators*

Generates random numbers with uniform distribution.

Syntax

```
status = vsRngUniform( method, stream, n, r, a, b );
```

```
status = vdRngUniform( method, stream, n, r, a, b );
```

Include Files

- `mk1.h`

Description

The `vRngUniform` function generates random numbers uniformly distributed over the interval $[a, b)$, where a, b are the left and right bounds of the interval, respectively, and $a, b \in \mathbb{R}$; $a < b$.

The probability density function is given by:

$$f_{a,b}(x) = \begin{cases} \frac{1}{b-a}, & x \in [a, b) \\ 0, & x \notin [a, b) \end{cases}, \quad -\infty < x < +\infty$$

The cumulative distribution function is as follows:

$$f_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{x-a}{b-a}, & a \leq x < b, \\ 1, & x \geq b \end{cases}, \quad -\infty < x < +\infty.$$

Product and Performance Information

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

Product and Performance Information

Notice revision #20201201

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method; the specific values are as follows: VSL_RNG_METHOD_UNIFORM_STD VSL_RNG_METHOD_UNIFORM_STD_ACCURATE Standard method.
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated.
<i>a</i>	const float for vsRngUniform const double for vdRngUniform	Left bound a.
<i>b</i>	const float for vsRngUniform const double for vdRngUniform	Right bound b.

Output Parameters

Name	Type	Description
<i>r</i>	float* for vsRngUniform double* for vdRngUniform	Vector of <i>n</i> random numbers uniformly distributed over the interval $[a, b)$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED

ARS-5 random number generator is not supported on the CPU running the application.

vRngGaussian

Generates normally distributed random numbers.

Syntax

```
status = vsRngGaussian( method, stream, n, r, a, sigma );
```

```
status = vdRngGaussian( method, stream, n, r, a, sigma );
```

Include Files

- mkl.h

Description

The `vRngGaussian` function generates random numbers with normal (Gaussian) distribution with mean value a and standard deviation σ , where

$a, \sigma \in \mathbb{R} ; \sigma > 0.$

The probability density function is given by:

$$f_{a, \sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-a)^2}{2\sigma^2}\right), \quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a, \sigma}(x) = \int_{-\infty}^x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-a)^2}{2\sigma^2}\right) dy, \quad -\infty < x < +\infty.$$

The cumulative distribution function $F_{a, \sigma}(x)$ can be expressed in terms of standard normal distribution $\Phi(x)$ as

$$F_{a, \sigma}(x) = \Phi((x-a)/\sigma)$$

Product and Performance Information

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

Notice revision #20201201

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: <div> <div>VSL_RNG_METHOD_GAUSSIAN_BOXMULLER</div> <div>VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2</div> <div>VSL_RNG_METHOD_GAUSSIAN_ICDF</div> </div> See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table "Values of <method> in method parameter"
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated.
<i>a</i>	const float for vsRngGaussian const double for vdRngGaussian	Mean value <i>a</i> .
<i>sigma</i>	const float for vsRngGaussian const double for vdRngGaussian	Standard deviation σ .

Output Parameters

Name	Type	Description
<i>r</i>	float* for vsRngGaussian double* for vdRngGaussian	Vector of <i>n</i> normally distributed random numbers.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGaussianMV

Generates random numbers from multivariate normal distribution.

Syntax

```
status = vsRngGaussianMV( method, stream, n, r, dimen, mstorage, a, t );
status = vdRngGaussianMV( method, stream, n, r, dimen, mstorage, a, t );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2 VSL_RNG_METHOD_GAUSSIANMV_ICDF See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table "Values of <method> in method parameter"
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of d -dimensional vectors to be generated
<i>dimen</i>	const MKL_INT	Dimension d ($d \geq 1$) of output random vectors
<i>mstorage</i>	const MKL_INT	Matrix storage scheme for lower triangular matrix T . The routine supports three matrix storage schemes: <ul style="list-style-type: none"> • VSL_MATRIX_STORAGE_FULL— all $d \times d$ elements of the matrix T are passed, however, only the lower triangle part is actually used in the routine.

Name	Type	Description
		<ul style="list-style-type: none"> VSL_MATRIX_STORAGE_PACKED— lower triangle elements of T are packed by rows into a one-dimensional array. VSL_MATRIX_STORAGE_DIAGONAL— only diagonal elements of T are passed.
a	const float* for vsRngGaussianMV const double* for vdRngGaussianMV	Mean vector a of dimension d
t	const float* for vsRngGaussianMV const double* for vdRngGaussianMV	Elements of the lower triangular matrix passed according to the matrix T storage scheme $mstorage$.

Output Parameters

Name	Type	Description
r	float* for vsRngGaussianMV double* for vdRngGaussianMV	Array of n random vectors of dimension $dimen$

Description

The `vsRngGaussianMV` function generates random numbers with d -variate normal (Gaussian) distribution with mean value a and variance-covariance matrix C , where $a \in \mathbb{R}^d$; C is a $d \times d$ symmetric positive-definite matrix.

The probability density function is given by:

$$f_{a,C}(x) = \frac{1}{\sqrt{\det(2\pi C)}} \exp(-1/2(x-a)^T C^{-1}(x-a)),$$

where $x \in \mathbb{R}^d$.

Matrix C can be represented as $C = TT^T$, where T is a lower triangular matrix - Cholesky factor of C .

Instead of variance-covariance matrix C the generation routines require Cholesky factor of C in input. To compute Cholesky factor of matrix C , the user may call Intel® oneAPI Math Kernel Library (oneMKL) LAPACK routines for matrix factorization: `?potrf` or `?pptrf` for `vsRngGaussianMV`/`vsrnggaussianmv` routines (? means either s or d for single and double precision respectively). See [Application Notes](#) for more details.

Product and Performance Information

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

Notice revision #20201201

Application Notes

Since matrices are stored in Fortran by columns, while in C they are stored by rows, the usage of Intel® oneAPI Math Kernel Library (oneMKL) factorization routines (assuming Fortran matrices storage) in combination with multivariate normal RNG (assuming C matrix storage) is slightly different in C and Fortran. The following tables help in using these routines in C and Fortran. For further information please refer to the appropriate VS example file.

Using Cholesky Factorization Routines in C

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Routine	UPLO Parameter in Factorization Routine	Result of Factorization as Input Argument for RNG
VSL_MATRIX_STORAGE_FULL	C in C two-dimensional array	spotrf for vsRngGaussianMV dpotrf for vdRngGaussianMV	'U'	Lower triangle of T . Upper triangle is not used.
VSL_MATRIX_STORAGE_PACKED	Lower triangle of C packed by columns into one-dimensional array	spptrf for vsRngGaussianMV dpptrf for vdRngGaussianMV	'L'	Lower triangle of T packed by columns into a one-dimensional array.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngExponential

Generates exponentially distributed random numbers.

Syntax

```
status = vsRngExponential( method, stream, n, r, a, beta );
```

```
status = vdRngExponential( method, stream, n, r, a, beta );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_EXPONENTIAL_ICDF VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE Inverse cumulative distribution function method
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>a</i>	const float for vsRngExponential const double for vdRngExponential	Displacement <i>a</i>
<i>beta</i>	const float for vsRngExponential const double for vdRngExponential	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	float* for vsRngExponential double* for vdRngExponential	Vector of <i>n</i> exponentially distributed random numbers

Description

The `vsRngExponential` function generates random numbers with exponential distribution that has displacement *a* and scalefactor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta} \exp(-(x-a)/\beta), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp(-(x-a)/\beta), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngLaplace

Generates random numbers with Laplace distribution.

Syntax

```
status = vsRngLaplace( method, stream, n, r, a, beta );
status = vdRngLaplace( method, stream, n, r, a, beta );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: <code>VSL_RNG_METHOD_LAPLACE_ICDF</code> Inverse cumulative distribution function method
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure

Name	Type	Description
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>a</i>	const float for vsRngLaplace const double for vdRngLaplace	Mean value <i>a</i>
<i>beta</i>	const float for vsRngLaplace const double for vdRngLaplace	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	float* for vsRngLaplace double* for vdRngLaplace	Vector of <i>n</i> Laplace distributed random numbers

Description

The `vsRngLaplace` function generates random numbers with Laplace distribution with mean value (or average) *a* and scalefactor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$. The scalefactor value determines the standard deviation as

$$\sigma = \beta\sqrt{2}$$

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\sqrt{2\beta}} \exp\left(-\frac{|x-a|}{\beta}\right), -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} \frac{1}{2} \exp\left(-\frac{|x-a|}{\beta}\right), & x \geq a \\ 1 - \frac{1}{2} \exp\left(-\frac{|x-a|}{\beta}\right), & x < a \end{cases}, -\infty < x < +\infty.$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
-----------------------------	--

VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vrngWeibull

Generates Weibull distributed random numbers.

Syntax

```
status = vsRngWeibull( method, stream, n, r, alpha, a, beta );
```

```
status = vdRngWeibull( method, stream, n, r, alpha, a, beta );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_WEIBULL_ICDF VSL_RNG_METHOD_WEIBULL_ICDF_ACCURATE Inverse cumulative distribution function method
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>alpha</i>	const float for vsRngWeibull const double for vdRngWeibull	Shape α .
<i>a</i>	const float for vsRngWeibull const double for vdRngWeibull	Displacement a
<i>beta</i>	const float for vsRngWeibull	Scalefactor β .

Name	Type	Description
------	------	-------------

	const double for vdRngWeibull	
--	----------------------------------	--

Output Parameters

Name	Type	Description
------	------	-------------

r	float* for vsRngWeibull double* for vdRngWeibull	Vector of n Weibull distributed random numbers
-----	---	--

Description

The `vdRngWeibull` function generates Weibull distributed random numbers with displacement a , scalefactor β , and shape α , where $\alpha, \beta, a \in \mathbb{R}$; $\alpha > 0, \beta > 0$.

The probability density function is given by:

$$f_{a,\alpha,\beta}(x) = \begin{cases} \frac{\alpha}{\beta^\alpha} (x - a)^{\alpha-1} \exp\left(-\left(\frac{x - a}{\beta}\right)^\alpha\right), & x \geq a \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\alpha,\beta}(x) = \begin{cases} 1 - \exp\left(-\left(\frac{x - a}{\beta}\right)^\alpha\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK

Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR

stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM

stream is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.

VSL_RNG_ERROR_NO_NUMBERS

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngCauchy

Generates Cauchy distributed random values.

Syntax

```
status = vsRngCauchy( method, stream, n, r, a, beta );
```

```
status = vdRngCauchy( method, stream, n, r, a, beta );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_CAUCHY_ICDF Inverse cumulative distribution function method
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>a</i>	const float for vsRngCauchy const double for vdRngCauchy	Displacement a .
<i>beta</i>	const float for vsRngCauchy const double for vdRngCauchy	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	float* for vsRngCauchy double* for vdRngCauchy	Vector of n Cauchy distributed random numbers

Description

The function generates Cauchy distributed random numbers with displacement a and scalefactor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\pi\beta \left(1 + \left(\frac{x-a}{\beta} \right)^2 \right)}, \quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \frac{1}{2} + \frac{1}{\pi} \arctan \left(\frac{x-a}{\beta} \right), \quad -\infty < x < +\infty.$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngRayleigh

Generates Rayleigh distributed random values.

Syntax

```
status = vsRngRayleigh( method, stream, n, r, a, beta );
status = vdRngRayleigh( method, stream, n, r, a, beta );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_RAYLEIGH_ICDF VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE Inverse cumulative distribution function method
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>a</i>	const float for vsRngRayleigh const double for vdRngRayleigh	Displacement <i>a</i>
<i>beta</i>	const float for vsRngRayleigh const double for vdRngRayleigh	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	float* for vsRngRayleigh double* for vdRngRayleigh	Vector of <i>n</i> Rayleigh distributed random numbers

Description

The `vsRngRayleigh` function generates Rayleigh distributed random numbers with displacement *a* and scalefactor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$.

The Rayleigh distribution is a special case of the [Weibull](#) distribution, where the shape parameter $\alpha = 2$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x-a)}{\beta^2} \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp\left(-\frac{(x-a)^2}{\beta^2}\right), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngLognormal

Generates lognormally distributed random numbers.

Syntax

```
status = vsRngLognormal( method, stream, n, r, a, sigma, b, beta );
status = vdRngLognormal( method, stream, n, r, a, sigma, b, beta );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2 VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE Box Muller 2 based method VSL_RNG_METHOD_LOGNORMAL_ICDF VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCURATE Inverse cumulative distribution function based method

Name	Type	Description
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>a</i>	const float for vsRngLognormal const double for vdRngLognormal	Average <i>a</i> of the subject normal distribution
<i>sigma</i>	const float for vsRngLognormal const double for vdRngLognormal	Standard deviation σ of the subject normal distribution
<i>b</i>	const float for vsRngLognormal const double for vdRngLognormal	Displacement <i>b</i>
<i>beta</i>	const float for vsRngLognormal const double for vdRngLognormal	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	float* for vsRngLognormal double* for vdRngLognormal	Vector of <i>n</i> lognormally distributed random numbers

Description

The `vsRngLognormal` function generates lognormally distributed random numbers with average of distribution *a* and standard deviation σ of subject normal distribution, displacement *b*, and scalefactor β , where *a*, σ , *b*, $\beta \in \mathbb{R}$; $\sigma > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{a,\sigma,b,\beta}(x) = \begin{cases} \frac{1}{\sigma(x-b)\sqrt{2\pi}} \exp\left(-\frac{[\ln((x-b)/\beta) - a]^2}{2\sigma^2}\right), & x > b \\ 0, & x \leq b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\sigma,b,\beta}(x) = \begin{cases} \Phi(\ln((x-b)/\beta) - a)/\sigma), & x > b \\ 0, & x \leq b \end{cases}$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGumbel

Generates Gumbel distributed random values.

Syntax

```
status = vsRngGumbel( method, stream, n, r, a, beta );
status = vdRngGumbel( method, stream, n, r, a, beta );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: <code>VSL_RNG_METHOD_GUMBEL_ICDF</code> Inverse cumulative distribution function method
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>a</i>	const float for vsRngGumbel	Displacement <i>a</i> .

Name	Type	Description
------	------	-------------

	<code>const double for vdRngGumbel</code>	
<i>beta</i>	<code>const float for vsRngGumbel</code>	Scalefactor β .
	<code>const double for vdRngGumbel</code>	

Output Parameters

Name	Type	Description
------	------	-------------

<i>r</i>	<code>float* for vsRngGumbel</code> <code>double* for vdRngGumbel</code>	Vector of n random numbers with Gumbel distribution
----------	---	---

Description

The `vdRngGumbel` function generates Gumbel distributed random numbers with displacement a and scalefactor β , where $a, \beta \in \mathbb{R}$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \left\{ \frac{1}{\beta} \exp\left(\frac{x-a}{\beta}\right) \exp(-\exp((x-a)/\beta)), -\infty < x < +\infty \right.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = 1 - \exp(-\exp((x-a)/\beta)), -\infty < x < +\infty$$

Product and Performance Information

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

Notice revision #20201201

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<i>stream</i> is a <code>NULL</code> pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is not a valid random stream.
<code>VSL_RNG_ERROR_BAD_UPDATE</code>	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
<code>VSL_RNG_ERROR_NO_NUMBERS</code>	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
<code>VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED</code>	Period of the generator has been exceeded.
<code>VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED</code>	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED

ARS-5 random number generator is not supported on the CPU running the application.

vRngGamma

Generates gamma distributed random values.

Syntax

```
status = vsRngGamma( method, stream, n, r, alpha, a, beta );
```

```
status = vdRngGamma( method, stream, n, r, alpha, a, beta );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_GAMMA_GNORM VSL_RNG_METHOD_GAMMA_GNORM_ACCURATE Acceptance/rejection method using random numbers with Gaussian distribution. See brief description of the method GNORM in Table "Values of <method> in method parameter"
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>alpha</i>	const float for vsRngGamma const double for vdRngGamma	Shape α .
<i>a</i>	const float for vsRngGamma const double for vdRngGamma	Displacement a .
<i>beta</i>	const float for vsRngGamma const double for vdRngGamma	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	float* for vsRngGamma double* for vdRngGamma	Vector of n random numbers with gamma distribution

Description

The `vsRngGamma` function generates random numbers with gamma distribution that has shape parameter α , displacement a , and scale parameter β , where α, β , and $a \in \mathbb{R}$; $\alpha > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{\alpha,a,\beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^\alpha} (x-a)^{\alpha-1} e^{-(x-a)/\beta}, & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

where $\Gamma(\alpha)$ is the complete gamma function.

The cumulative distribution function is as follows:

$$F_{\alpha,a,\beta}(x) = \begin{cases} \int_a^x \frac{1}{\Gamma(\alpha)\beta^\alpha} (y-a)^{\alpha-1} e^{-(y-a)/\beta} dy, & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngBeta

Generates beta distributed random values.

Syntax

```
status = vsRngBeta( method, stream, n, r, p, q, a, beta );
```

```
status = vdRngBeta( method, stream, n, r, p, q, a, beta );
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>method</i>	<code>const MKL_INT</code>	Generation method. The specific values are as follows: <div>VSL_RNG_METHOD_BETA_CJA</div> <div>VSL_RNG_METHOD_BETA_CJA_ACCURATE</div> See brief description of the method CJA in Table "Values of <method> in method parameter"
<i>stream</i>	<code>VSLStreamStatePtr</code>	Pointer to the stream state structure
<i>n</i>	<code>const MKL_INT</code>	Number of random values to be generated
<i>p</i>	<code>const float</code> for <code>vsRngBeta</code> <code>const double</code> for <code>vdRngBeta</code>	Shape <i>p</i>
<i>q</i>	<code>const float</code> for <code>vsRngBeta</code> <code>const double</code> for <code>vdRngBeta</code>	Shape <i>q</i>
<i>a</i>	<code>const float</code> for <code>vsRngBeta</code> <code>const double</code> for <code>vdRngBeta</code>	Displacement <i>a</i> .
<i>beta</i>	<code>const float</code> for <code>vsRngBeta</code> <code>const double</code> for <code>vdRngBeta</code>	Scale factor β .

Output Parameters

Name	Type	Description
<i>r</i>	<code>float*</code> for <code>vsRngBeta</code> <code>double*</code> for <code>vdRngBeta</code>	Vector of <i>n</i> random numbers with beta distribution

Description

The `vsRngBeta` function generates random numbers with beta distribution that has shape parameters *p* and *q*, displacement *a*, and scale parameter β , where *p*, *q*, *a*, and $\beta \in \mathbb{R}$; *p* > 0, *q* > 0, β > 0.

The probability density function is given by:

$$f_{p,q,a,\beta}(x) = \begin{cases} \frac{1}{B(p,q)\beta^{p+q-1}} (x-a)^{p-1} (\beta+a-x)^{q-1}, & a \leq x < a + \beta \\ 0, & x < a, x \geq a + \beta \end{cases}, -\infty < x < \infty$$

where $B(p, q)$ is the complete beta function.

The cumulative distribution function is as follows:

$$F_{p,q,a,\beta}(x) = \begin{cases} 0, & x < a \\ \int_a^x \frac{1}{B(p,q)\beta^{p+q-1}} (y-a)^{p-1} (\beta+a-y)^{q-1} dy, & a \leq x < a + \beta, -\infty < x \\ 1, & x \geq a + \beta \end{cases}$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngChiSquare

Generates chi-square distributed random values.

Syntax

```
status = vsRngChiSquare( method, stream, n, r, v );
```

```
status = vdRngChiSquare( method, stream, n, r, v );
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific value is: <code>VSL_RNG_METHOD_CHISQUARE_CHI2GAMMA</code>

Name	Type	Description
		For a description of VSL_RNG_METHOD_CHISQUARE_CHI2GAMMA, see Random Number Generators Naming Conventions .
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>v</i>	const MKL_INT	Degrees of freedom

Output Parameters

Name	Type	Description
<i>r</i>	float* for vsRngChiSquare double* for vdRngChiSquare	Vector of <i>n</i> random numbers with chi-square distribution

Description

The `vsRngChiSquare` function generates random numbers with chi-square distribution and *v* degrees of freedom, $v \in \mathbb{N}$, $v > 0$.

The probability density function is:

$$f_v(x) = \begin{cases} \frac{x^{\frac{v-2}{2}} e^{-\frac{x}{2}}}{2^{v/2} \Gamma(\frac{v}{2})}, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

The cumulative distribution function is:

$$F_v(x) = \begin{cases} \int_0^x \frac{y^{\frac{v-2}{2}} e^{-\frac{y}{2}}}{2^{v/2} \Gamma(\frac{v}{2})} dy, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

Discrete Distributions

This section describes routines for generating random numbers with discrete distribution.

`vRngUniform` *Discrete Distribution Generators*

Generates random numbers uniformly distributed over the interval $[a, b)$.

Syntax

```
status = viRngUniform( method, stream, n, r, a, b );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>method</code>	<code>const MKL_INT</code>	Generation method; the specific value is as follows: <code>VSL_RNG_METHOD_UNIFORM_STD</code> Standard method. Currently there is only one method for this distribution generator.
<code>stream</code>	<code>VSLStreamStatePtr</code>	Pointer to the stream state structure
<code>n</code>	<code>const MKL_INT</code>	Number of random values to be generated
<code>a</code>	<code>const int</code>	Left interval bound a
<code>b</code>	<code>const int</code>	Right interval bound b

Output Parameters

Name	Type	Description
<code>r</code>	<code>int*</code>	Vector of n random numbers uniformly distributed over the interval $[a, b)$

Description

The `vRngUniform` function generates random numbers uniformly distributed over the interval $[a, b)$, where a, b are the left and right bounds of the interval respectively, and $a, b \in \mathbb{Z}; a < b$.

The probability distribution is given by:

$$P(X = k) = \frac{1}{b - a}, k \in \{a, a + 1, \dots, b - 1\}.$$

The cumulative distribution function is as follows:

$$F_{a,b}(X) = \begin{cases} 0, & x < a \\ \frac{\lfloor x - a + 1 \rfloor}{b - a}, & a \leq x < b, x \in R. \\ 1, & x \geq b \end{cases}$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngUniformBits

Generates bits of underlying BRNG integer recurrence.

Syntax

```
status = viRngUniformBits( method, stream, n, r );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method; the specific value is <code>VSL_RNG_METHOD_UNIFORMBITS_STD</code>
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated

Output Parameters

Name	Type	Description
<i>r</i>	unsigned int*	Vector of <i>n</i> random integer numbers. If the <i>stream</i> was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of <i>r</i> respectively. The number of bytes occupied by each integer is contained in the field <i>WordSize</i> of the structure <code>VSLBRngProperties</code> . The total number of bits that are actually used to store the value are contained in the field <i>NBits</i> of the same structure. See Advanced Service Routines for a more detailed discussion of <code>VSLBRngProperties</code> .

Description

The `vRngUniformBits` function generates integer random values with uniform bit distribution. The generators of uniformly distributed numbers can be represented as recurrence relations over integer values in modular arithmetic. Apparently, each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated. For example, a well known drawback of linear congruential generators is that lower bits are less random than higher bits (for example, see [Knuth81]). For this reason, care should be taken when using this function. Typically, in a 32-bit LCG only 24 higher bits of an integer value can be considered random. See [VS Notes](#) for details.

Product and Performance Information

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

Notice revision #20201201

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<i>stream</i> is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is not a valid random stream.
<code>VSL_RNG_ERROR_BAD_UPDATE</code>	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
<code>VSL_RNG_ERROR_NO_NUMBERS</code>	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngUniformBits32

Generates uniformly distributed bits in 32-bit chunks.

Syntax

```
status = viRngUniformBits32( method, stream, n, r );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method; the specific value is VSL_RNG_METHOD_UNIFORMBITS32_STD
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated

Output Parameters

Name	Type	Description
<i>r</i>	unsigned int*	Vector of <i>n</i> 32-bit random integer numbers with uniform bit distribution.

Description

The `vRngUniformBits32` function generates uniformly distributed bits in 32-bit chunks. Unlike `vRngUniformBits`, which provides the output of underlying integer recurrence and does not guarantee uniform distribution across bits, `vRngUniformBits32` is designed to ensure each bit in the 32-bit chunk is uniformly distributed. See [VS Notes](#) for details.

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngUniformBits64

Generates uniformly distributed bits in 64-bit chunks.

Syntax

```
status = viRngUniformBits64( method, stream, n, r );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method; the specific value is VSL_RNG_METHOD_UNIFORMBITS64_STD
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated

Output Parameters

Name	Type	Description
<i>r</i>	unsigned MKL_INT64*	Vector of <i>n</i> 64-bit random integer numbers with uniform bit distribution.

Description

The `vRngUniformBits64` function generates uniformly distributed bits in 64-bit chunks. Unlike `vRngUniformBits`, which provides the output of underlying integer recurrence and does not guarantee uniform distribution across bits, `vRngUniformBits64` is designed to ensure each bit in the 64-bit chunk is uniformly distributed. See [VS Notes](#) for details.

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
-----------------------------	--

VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngBernoulli

Generates Bernoulli distributed random values.

Syntax

```
status = viRngBernoulli( method, stream, n, r, p );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific value is as follows: VSL_RNG_METHOD_BERNOULLI_ICDF Inverse cumulative distribution function method.
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>p</i>	const double	Success probability <i>p</i> of a trial

Output Parameters

Name	Type	Description
<i>r</i>	int*	Vector of <i>n</i> Bernoulli distributed random values

Description

The *vRngBernoulli* function generates Bernoulli distributed random numbers with probability *p* of a single trial success, where

$$p \in \mathbb{R}; 0 \leq p \leq 1.$$

A variate is called Bernoulli distributed, if after a trial it is equal to 1 with probability of success *p*, and to 0 with probability 1 - *p*.

The probability distribution is given by:

$$P(X = 1) = p$$

$$P(X = 0) = 1 - p$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - p, & 0 \leq x < 1, x \in R. \\ 1, & x \geq 1 \end{cases}$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngGeometric
Generates geometrically distributed random values.

Syntax

status = viRngGeometric(*method*, *stream*, *n*, *r*, *p*);

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific value is as follows: VSL_RNG_METHOD_GEOMETRIC_ICDF

Name	Type	Description
		Inverse cumulative distribution function method.
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>p</i>	const double	Success probability <i>p</i> of a trial

Output Parameters

Name	Type	Description
<i>r</i>	int*	Vector of <i>n</i> geometrically distributed random values

Description

The `vRngGeometric` function generates geometrically distributed random numbers with probability *p* of a single trial success, where $p \in \mathbb{R}; 0 < p < 1$.

A geometrically distributed variate represents the number of independent Bernoulli trials preceding the first success. The probability of a single Bernoulli trial success is *p*.

The probability distribution is given by:

$$P(X = k) = p \cdot (1 - p)^k, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - (1 - p)^{\lfloor x+1 \rfloor}, & 0 \leq x \end{cases} \quad x \in \mathbb{R}.$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED ARS-5 random number generator is not supported on the CPU running the application.

vRngBinomial

Generates binomially distributed random numbers.

Syntax

```
status = viRngBinomial( method, stream, n, r, ntrial, p );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific value is as follows: VSL_RNG_METHOD_BINOMIAL_BTPE See brief description of the BTPE method in Table "Values of <method> in method parameter" .
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>ntrial</i>	const int	Number of independent trials m
<i>p</i>	const double	Success probability p of a single trial

Output Parameters

Name	Type	Description
<i>r</i>	int*	Vector of n binomially distributed random values

Description

The `viRngBinomial` function generates binomially distributed random numbers with number of independent Bernoulli trials m , and with probability p of a single trial success, where $p \in R$; $0 \leq p \leq 1$, $m \in N$.

A binomially distributed variate represents the number of successes in m independent Bernoulli trials with probability of a single trial success p .

The probability distribution is given by:

$$P(X = k) = C_m^k p^k (1 - p)^{m-k}, k \in \{0, 1, \dots, m\}.$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, & x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1-p)^{m-k}, & 0 \leq x < m, x \in \mathbb{R} \\ 1, & x \geq m \end{cases}$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngHypergeometric

Generates hypergeometrically distributed random values.

Syntax

```
status = viRngHypergeometric( method, stream, n, r, l, s, m );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific value is as follows: VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE See brief description of the H2PE method in Table "Values of <method> in method parameter"
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>l</i>	const int	Lot size l
<i>s</i>	const int	Size of sampling without replacement s
<i>m</i>	const int	Number of marked elements m

Output Parameters

Name	Type	Description
<i>r</i>	int*	Vector of n hypergeometrically distributed random values

Description

The `vRngHypergeometric` function generates hypergeometrically distributed random values with lot size l , size of sampling s , and number of marked elements in the lot m , where $l, m, s \in \mathbb{N} \setminus \{0\}$; $l \geq \max(s, m)$.

Consider a lot of l elements comprising m "marked" and $l-m$ "unmarked" elements. A trial sampling without replacement of exactly s elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of s elements contains exactly k marked elements.

The probability distribution is given by:)

$$P(X = k) = \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}$$

, $k \in \{\max(0, s + m - l), \dots, \min(s, m)\}$

The cumulative distribution function is as follows:

$$F_{l,s,m}(x) = \begin{cases} 0, & x < \max(0, s + m - l) \\ \sum_{k=\max(0, s+m-l)}^{\lfloor x \rfloor} \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, & \max(0, s + m - l) \leq x \leq \min(s, m) \\ 1, & x > \min(s, m) \end{cases}$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngPoisson

Generates Poisson distributed random values.

Syntax

```
status = viRngPoisson( method, stream, n, r, lambda );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_POISSON_PTPE VSL_RNG_METHOD_POISSON_POISNORM See brief description of the PTPE and POISNORM methods in Table "Values of <method> in method parameter" .
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>lambda</i>	const double	Distribution parameter λ .

Output Parameters

Name	Type	Description
<i>r</i>	int*	Vector of <i>n</i> Poisson distributed random values

Description

The `vRng"Poisson` function generates Poisson distributed random numbers with distribution parameter λ , where $\lambda \in \mathbb{R}$; $\lambda > 0$.

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!},$$

$k \in \{0, 1, 2, \dots\}$.

The cumulative distribution function is as follows:

$$F_{\lambda}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in \mathbb{R}$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> n_{\max}$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngPoissonV

Generates Poisson distributed random values with varying mean.

Syntax

```
status = viRngPoissonV( method, stream, n, r, lambda );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific value is as follows: VSL_RNG_METHOD_POISSONV_POISNORM See brief description of the POISNORM method in Table "Values of <method> in method parameter"
<i>stream</i>	VSLStreamStatePtr	Pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>lambda</i>	const double*	Array of <i>n</i> distribution parameters λ_i .

Output Parameters

Name	Type	Description
<i>r</i>	int*	Vector of <i>n</i> Poisson distributed random values

Description

The `vRngPoissonV` function generates *n* Poisson distributed random numbers $x_i (i = 1, \dots, n)$ with distribution parameter λ_i , where $\lambda_i \in \mathbb{R}; \lambda_i > 0$.

The probability distribution is given by:

$$P(X_i = k) = \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_{\lambda_i}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k e^{-\lambda_i}}{k!}, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngNegBinomial

Generates random numbers with negative binomial distribution.

Syntax

```
status = viRngNegbinomial( method, stream, n, r, a, p );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	const MKL_INT	Generation method. The specific value is: VSL_RNG_METHOD_NEGBINOMIAL_NBAR See brief description of the NBAR method in Table "Values of <method> in method parameter"
<i>stream</i>	VSLStreamStatePtr	pointer to the stream state structure
<i>n</i>	const MKL_INT	Number of random values to be generated
<i>a</i>	const double	The first distribution parameter <i>a</i>
<i>p</i>	const double	The second distribution parameter <i>p</i>

Output Parameters

Name	Type	Description
<i>r</i>	int*	Vector of <i>n</i> random values with negative binomial distribution.

Description

The `vRngNegBinomial` function generates random numbers with negative binomial distribution and distribution parameters *a* and *p*, where $p, a \in R$; $0 < p < 1$; $a > 0$.

If the first distribution parameter $a \in N$, this distribution is the same as Pascal distribution. If $a \in N$, the distribution can be interpreted as the expected time of *a*-th success in a sequence of Bernoulli trials, when the probability of success is *p*.

The probability distribution is given by:

$$P(X = k) = C_{a+k-1}^k p^a (1 - p)^k, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_{a,p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C_{a+k-1}^k p^a (1 - p)^k, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_ARS5_NOT_SUPPORTED	ARS-5 random number generator is not supported on the CPU running the application.

vRngMultinomial

Generates multinomially distributed random numbers.

Syntax

```
status = viRngMultinomial( method, stream, n, r, ntrial, k, p );
```

Include Files

- mkl.h

Input Parameters

<i>method</i>	const MKL_INT Generation method. The specific value is as follows: VSL_RNG_METHOD_MULTINOMIAL_MULTPOISSON
<i>stream</i>	VSLStreamStatePtr Pointer to the stream state structure.
<i>n</i>	const MKL_INT Number of random values to be generated.
<i>ntrial</i>	const int Number of independent trials m .
<i>k</i>	const int Number of possible outcomes.
<i>p</i>	const double* Probability vector of k possible outcomes.

Output Parameters

```

r                                     int*

```

Array of n random vectors of dimension k .

Description

The `vRngMultinomial` function generates multinomially distributed random numbers with m independent trials and k possible mutually exclusive outcomes, with corresponding probabilities p_i , where $p_i \in \mathbb{R}$; $0 \leq p_i \leq 1$, $m \in \mathbb{N}$, $k \in \mathbb{N}$.

The probability distribution is given by:

$$P(X_1 = x_1, \dots, X_k = x_k) = \frac{m!}{\prod_{i=1}^k x_i!} \prod_{i=1}^k p_i^{x_i}, \quad 0 \leq x_i \leq m, \sum_{i=1}^k x_i = m$$

Product and Performance Information

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

Notice revision #20201201

Return Values

VSL_ERROR_OK,	Indicates no error (execution was successful).
VSL_STATUS_OK	
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STR	<i>stream</i> is not a valid random stream.
EAM	
VSL_RNG_ERROR_BAD_UPD	A callback function for an abstract BRNG returns an invalid number of updated
ATE	entries in a buffer; that is, < 0 or $> n_{\max}$.
VSL_RNG_ERROR_NO_NUMB	A callback function for an abstract BRNG returns 0 as the number of updated
ERS	entries in a buffer.
VSL_RNG_ERROR_ARS5_NO	An ARS-5 random number generator is not supported on the CPU running the
T_SUPPORTED	application.
VSL_DISTR_MULTINOMIAL	Bad multinomial distribution probability array.
_BAD_PROBABILITY_ARRA	
Y	

Advanced Service Routines

This section describes service routines for registering a user-designed basic generator ([vslRegisterBrng](#)) and for obtaining properties of the previously registered basic generators ([vslGetBrngProperties](#)). See [VS Notes](#) ("Basic Generators" section of VS Structure chapter) for substantiation of the need for several basic generators including user-defined BRNGs.

Advanced Service Routine Data Types

The Advanced Service routines refer to a structure defining the properties of the basic generator.

This structure is described as follows:

```
typedef struct _VSLBRngProperties {
    int StreamStateSize;
    int NSeeds;
    int IncludesZero;
    int WordSize;
    int NBits;
    InitStreamPtr InitStream;
    sBRngPtr sBRng;
    dBRngPtr dBRng;
    iBRngPtr iBRng;
} VSLBRngProperties;
```

The following table provides brief descriptions of the fields engaged in the above structure:

Field Descriptions

Field	Short Description
StreamStateSize	The size, in bytes, of the stream state structure for a given basic generator.
NSeeds	The number of 32-bit initial conditions (seeds) necessary to initialize the stream state structure for a given basic generator.
IncludesZero	Flag value indicating whether the generator can produce a random 0.
WordSize	Machine word size, in bytes, used in integer-value computations. Possible values: 4, 8, and 16 for 32, 64, and 128-bit generators, respectively.
NBits	The number of bits required to represent a random value in integer arithmetic. Note that, for instance, 48-bit random values are stored to 64-bit (8 byte) memory locations. In this case, <code>wordsize/WordSize</code> is equal to 8 (number of bytes used to store the random value), while <code>nbits/NBits</code> contains the actual number of bits occupied by the value (in this example, 48).
InitStream	Contains the pointer to the initialization routine of a given basic generator.
sBRng	Contains the pointer to the basic generator of single precision real numbers uniformly distributed over the interval (a,b) (<code>float</code>).
dBRng	Contains the pointer to the basic generator of double precision real numbers uniformly distributed over the interval (a,b) (<code>double</code>).
iBRng	Contains the pointer to the basic generator of integer numbers with uniform bit distribution (<code>unsigned int</code>).

vslRegisterBrng

Registers user-defined basic generator.

Syntax

```
brng = vslRegisterBrng( &properties );
```

¹ A specific generator that permits operations over single bits and bit groups of random numbers.

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>properties</code>	<code>const VSLBRngProperties*</code>	Pointer to the structure containing properties of the basic generator to be registered

Output Parameters

Name	Type	Description
<code>brng</code>	<code>int</code>	Number (index) of the registered basic generator; used for identification. Negative values indicate the registration error.

Description

An example of a registration procedure can be found in the respective directory of the VS examples.

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_RNG_ERROR_BRNG_TABLE_FULL</code>	Registration cannot be completed due to lack of free entries in the table of registered BRNGs.
<code>VSL_RNG_ERROR_BAD_STREAM_STATE_SIZE</code>	Bad value in <code>StreamStateSize</code> field.
<code>VSL_RNG_ERROR_BAD_WORD_SIZE</code>	Bad value in <code>WordSize</code> field.
<code>VSL_RNG_ERROR_BAD_NSEEDS</code>	Bad value in <code>NSeeds</code> field.
<code>VSL_RNG_ERROR_BAD_NBITS</code>	Bad value in <code>NBits</code> field.
<code>VSL_ERROR_NULL_PTR</code>	At least one of the fields <code>iBrng</code> , <code>dBrng</code> , <code>sBrng</code> or <code>InitStream</code> is a NULL pointer.

vslGetBrngProperties

Returns structure with properties of a given basic generator.

Syntax

```
status = vslGetBrngProperties( brng, &properties );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>brng</i>	const int	Number (index) of the registered basic generator; used for identification. See specific values in Table "Values of <i>brng</i> parameter" . Negative values indicate the registration error.

Output Parameters

Name	Type	Description
<i>properties</i>	VSLBRngProperties*	Pointer to the structure containing properties of the generator with number <i>brng</i>

Description

The `vslGetBrngProperties` function returns a structure with properties of a given basic generator.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.

Formats for User-Designed Generators

To register a user-designed basic generator using `vslRegisterBrng` function, you need to pass the pointer *iBrng* to the integer-value implementation of the generator; the pointers *sBrng* and *dBrng* to the generator implementations for single and double precision values, respectively; and pass the pointer *InitStream* to the stream initialization routine. See recommendations below on defining such functions with input and output arguments. An example of the registration procedure for a user-designed generator can be found in the respective directory of VS examples.

The respective pointers are defined as follows:

```
typedef int(*InitStreamPtr)( int method, VSLStreamStatePtr stream, int n, const unsigned int
params[] );

typedef int(*sBrngPtr)( VSLStreamStatePtr stream, int n, float r[], float a, float b );

typedef int(*dBrngPtr)( VSLStreamStatePtr stream, int n, double r[], double a, double b );

typedef int(*iBrngPtr)( VSLStreamStatePtr stream, int n, unsigned int r[] );
```

InitStream

```
int MyBrngInitStream( int method, VSLStreamStatePtr stream, int n, const unsigned int params[] )
{
    /* Initialize the stream */
    ...
} /* MyBrngInitStream */
```

Description

The initialization routine of a user-designed generator must initialize *stream* according to the specified initialization *method*, initial conditions *params* and the argument *n*. The value of *method* determines the initialization method to be used.

- If *method* is equal to 1, the initialization is by the standard generation method, which must be supported by all basic generators. In this case the function assumes that the *stream* structure was not previously initialized. The value of *n* is used as the actual number of 32-bit values passed as initial conditions through *params*. Note, that the situation when the actual number of initial conditions passed to the function is not sufficient to initialize the generator is not an error. Whenever it occurs, the basic generator must initialize the missing conditions using default settings.
- If *method* is equal to 2, the generation is by the leapfrog method, where *n* specifies the number of computational nodes (independent streams). Here the function assumes that the *stream* was previously initialized by the standard generation method. In this case *params* contains only one element, which identifies the computational node. If the generator does not support the leapfrog method, the function must return the error code `VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED`.
- If *method* is equal to 3, the generation is by the block-splitting method. Same as above, the *stream* is assumed to be previously initialized by the standard generation method; *params* is not used, *n* identifies the number of skipped elements. If the generator does not support the block-splitting method, the function must return the error code `VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED`.
- If *method* is equal to 4, the generation is by the advanced block-splitting method. The stream is assumed to be previously initialized by the standard generation method; *params* is converted to `MKL_UINT64[]` and *n* is used as actual number of 64-bit values in *params*. If the generator does not support the advanced block-splitting method, the function must return the error code `VSL_RNG_ERROR_SKIPAHEAD_EX_UNSUPPORTED`.

For a more detailed description of the leapfrog and the block-splitting methods, refer to the description of [vslLeapfrogStream](#), [vslSkipAheadStream](#), and [vslSkipAheadStreamEx](#), respectively.

Stream state structure is individual for every generator. However, each structure has a number of fields that are the same for all the generators:

```
typedef struct
{
    unsigned int Reserved1[2];
    unsigned int Reserved2[2];
    [fields specific for the given generator]
} MyStreamState;
```

The fields *Reserved1* and *Reserved2* are reserved for private needs only, and must not be modified by the user. When including specific fields into the structure, follow the rules below:

- The fields must fully describe the current state of the generator. For example, the state of a linear congruential generator can be identified by only one initial condition;
- If the generator can use both the leapfrog and the block-splitting methods, additional fields should be introduced to identify the independent streams. For example, in $LCG(a, c, m)$, apart from the initial conditions, two more fields should be specified: the value of the multiplier a^k and the value of the increment $(a^k - 1)c / (a - 1)$.

For a more detailed discussion, refer to [Knuth81], and [Gentle98]. An example of the registration procedure can be found in the respective directory of VS examples.

iBRng

```
int iMyBrng( VSLStreamStatePtr stream, int n, unsigned int r[] )
{
    int i; /* Loop variable */
    /* Generating integer random numbers */
    /* Pay attention to word size needed to
       store only random number */
    for( i = 0; i < n; i++)
    {
        r[i] = ...;
```

```

}
/* Update stream state */
...
return errcode;
} /* iMyBrng */

```

NOTE

When using 64 and 128-bit generators, consider digit capacity to store the numbers to the random vector r correctly. For example, storing one 64-bit value requires two elements of r , the first to store the lower 32 bits and the second to store the higher 32 bits. Similarly, use 4 elements of r to store a 128-bit value.

sBRng

```

int sMyBrng( VSLStreamStatePtr stream, int n, float r[], float a, float b )
{
    int i; /* Loop variable */
    /* Generating float (a,b) random numbers */
    for ( i = 0; i < n; i++ )
    {
        r[i] = ...;
    }
    /* Update stream state */
    ...
    return errcode;
} /* sMyBrng */

```

dBRng

```

int dMyBrng( VSLStreamStatePtr stream, int n, double r[], double a, double b )
{
    int i; /* Loop variable */
    /* Generating double (a,b) random numbers */
    for ( i = 0; i < n; i++ )
    {
        r[i] = ...;
    }
    /* Update stream state */
    ...
    return errcode;
} /* dMyBrng */

```

Convolution and Correlation

Intel® oneAPI Math Kernel Library (oneMKL) VS provides a set of routines intended to perform linear convolution and correlation transformations for single and double precision real and complex data.

For correct definition of implemented operations, see the [Mathematical Notation and Definitions](#).

The current implementation provides:

- Fourier algorithms for one-dimensional single and double precision real and complex data
- Fourier algorithms for multi-dimensional single and double precision real and complex data
- Direct algorithms for one-dimensional single and double precision real and complex data
- Direct algorithms for multi-dimensional single and double precision real and complex data

One-dimensional algorithms cover the following functions from the IBM* ESSL library:

```
SCONF, SCORF
```

```
SCOND, SCORD
```

```
SDCON, SDCOR
```

```
DDCON, DDCOR
```

```
SDDCON, SDDCOR.
```

Special wrappers are designed to simulate these ESSL functions. The wrappers are provided as sample sources:

```
${MKL}/examples/vslc/essl/vsl_wrappers
```

Additionally, you can browse the examples demonstrating the calculation of the ESSL functions through the wrappers:

```
${MKL}/examples/vslc/essl
```

The convolution and correlation API provides interfaces for Fortran 90 and C/89 languages. You can use the C89 interface with later versions of the C/C++.

Intel® oneAPI Math Kernel Library (oneMKL) provides the `mk1_vsl.h` header file. All header files are in the directory

```
${MKL}/include
```

The convolution and correlation API is implemented through task objects, or tasks. Task object is a data structure, or descriptor, which holds parameters that determine the specific convolution or correlation operation. Such parameters may be precision, type, and number of dimensions of user data, an identifier of the computation algorithm to be used, shapes of data arrays, and so on.

All the Intel® oneAPI Math Kernel Library (oneMKL) VS convolution and correlation routines process task objects in one way or another: either create a new task descriptor, change the parameter settings, compute mathematical results of the convolution or correlation using the stored parameters, or perform other operations. Accordingly, all routines are split into the following groups:

Task Constructors - routines that create a new task object descriptor and set up most common parameters.

Task Editors - routines that can set or modify some parameter settings in the existing task descriptor.

Task Execution Routines - compute results of the convolution or correlation operation over the actual input data, using the operation parameters held in the task descriptor.

Task Copy - routines used to make several copies of the task descriptor.

Task Destructors - routines that delete task objects and free the memory.

When the task is executed or copied for the first time, a special process runs which is called task commitment. During this process, consistency of task parameters is checked and the required work data are prepared. If the parameters are consistent, the task is tagged as committed successfully. The task remains committed until you edit its parameters. Hence, the task can be executed multiple times after a single commitment process. Since the task commitment process may include costly intermediate calculations such as preparation of Fourier transform of input data, launching the process only once can help speed up overall performance.

Convolution and Correlation Naming Conventions

The names of routines, types, and constants in the convolution and correlation API are case-sensitive and can contain both lowercase and uppercase characters (`vslsConvExec`).

The names of routines have the following structure:

```
vsl[datatype]{Conv|Corr}<base name>
```

where

- `vsl` is a prefix indicating that the routine belongs to Intel® MKL Vector Statistics.
- `[datatype]` is optional. If present, the symbol specifies the type of the input and output data and can be `s` (for single precision real type), `d` (for double precision real type), `c` (for single precision complex type), or `z` (for double precision complex type).
- `Conv` or `Corr` specifies whether the routine refers to convolution or correlation task, respectively.
- `<base name>` field specifies a particular functionality that the routine is designed for, for example, `NewTask`, `DeleteTask`.

Convolution and Correlation Data Types

All convolution or correlation routines use the following types for specifying data objects:

Type	Data Object
<code>VSLConvTaskPtr</code>	Pointer to a task descriptor for convolution
<code>VSLCorrTaskPtr</code>	Pointer to a task descriptor for correlation
<code>float</code>	Input/output user real data in single precision
<code>double</code>	Input/output user real data in double precision
<code>MKL_Complex8</code>	Input/output user complex data in single precision
<code>MKL_Complex16</code>	Input/output user complex data in double precision
<code>int</code>	All other data

Generic integer type (without specifying the byte size) is used for all integer data.

NOTE

The actual size of the generic integer type is platform-dependent. Before you compile your application, set an appropriate byte size for integers. See details in the 'Using the ILP64 Interface vs. LP64 Interface' section of the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

Convolution and Correlation Parameters

Basic parameters held by the task descriptor are assigned values when the task object is created, copied, or modified by task editors. Parameters of the correlation or convolution task are initially set up by task constructors when the task object is created. Parameter changes or additional settings are made by task editors. More parameters which define location of the data being convolved need to be specified when the task execution routine is invoked.

According to how the parameters are passed or assigned values, all of them can be categorized as either explicit (directly passed as routine parameters when a task object is created or executed) or optional (assigned some default or implicit values during task construction).

The following table lists all applicable parameters used in the Intel® oneAPI Math Kernel Library (oneMKL) convolution and correlation API.

Convolution and Correlation Task Parameters

Name	Category	Type	Default Value Label	Description
<code>job</code>	explicit	integer	Implied by the constructor name	Specifies whether the task relates to convolution or correlation

Name	Category	Type	Default Value Label	Description
<i>type</i>	explicit	integer	Implied by the constructor name	Specifies the type (real or complex) of the input/output data. Set to real in the current version.
<i>precision</i>	explicit	integer	Implied by the constructor name	Specifies precision (single or double) of the input/output data to be provided in arrays <i>x,y,z</i> .
<i>mode</i>	explicit	integer	None	Specifies whether the convolution/correlation computation should be done via Fourier transforms, or by a direct method, or by automatically choosing between the two. See SetMode for the list of named constants for this parameter.
<i>method</i>	optional	integer	"auto"	Hints at a particular computation method if several methods are available for the given <i>mode</i> . Setting this parameter to "auto" means that software will choose the best available method.
<i>internal_precision</i>	optional	integer	Set equal to the value of <i>precision</i>	Specifies precision of internal calculations. Can enforce double precision calculations even when input/output data are single precision. See SetInternalPrecision for the list of named constants for this parameter.
<i>dims</i>	explicit	integer	None	Specifies the rank (number of dimensions) of the user data provided in arrays <i>x,y,z</i> . Can be in the range from 1 to 7.
<i>x,y</i>	explicit	real arrays	None	Specify input data arrays. See Data Allocation for more information.
<i>z</i>	explicit	real array	None	Specifies output data array. See Data Allocation for more information.
<i>xshape, yshape, zshape</i>	explicit	integer arrays	None	Define shapes of the arrays <i>x, y, z</i> . See Data Allocation for more information.
<i>xstride, ystride, zstride</i>	explicit	integer arrays	None	Define strides within arrays <i>x, y, z</i> , that is specify the physical location of the input and output data in these arrays. See Data Allocation for more information.
<i>start</i>	optional	integer array	Undefined	Defines the first element of the mathematical result that will be stored to output array <i>z</i> . See SetStart and Data Allocation for more information.
<i>decimation</i>	optional	integer array	Undefined	Defines how to thin out the mathematical result that will be stored to output array <i>z</i> . See SetDecimation and Data Allocation for more information.

Users may pass the NULL pointer instead of either or all of the parameters *xstride*, *ystride*, or *zstride* for multi-dimensional calculations. In this case, the software assumes the dense data allocation for the arrays *x*, *y*, or *z* due to the Fortran-style "by columns" representation of multi-dimensional arrays.

Convolution and Correlation Task Status and Error Reporting

The task status is an integer value, which is zero if no error has been detected while processing the task, or a specific non-zero error code otherwise. Negative status values indicate errors, and positive values indicate warnings.

An error can be caused by invalid parameter values, a system fault like a memory allocation failure, or can be an internal error self-detected by the software.

Each task descriptor contains the current status of the task. When creating a task object, the constructor assigns the `VSL_STATUS_OK` status to the task. When processing the task afterwards, other routines such as editors or executors can change the task status if an error occurs and write a corresponding error code into the task status field.

Note that at the stage of creating a task or editing its parameters, the set of parameters may be inconsistent. The parameter consistency check is only performed during the task commitment operation, which is implicitly invoked before task execution or task copying. If an error is detected at this stage, task execution or task copying is terminated and the task descriptor saves the corresponding error code. Once an error occurs, any further attempts to process that task descriptor is terminated and the task keeps the same error code.

Normally, every convolution or correlation function (except `DeleteTask`) returns the status assigned to the task while performing the function operation.

The header files define symbolic names for the status codes. These names are defined as macros via the `#define` statements.

If there is no error, the `VSL_STATUS_OK` status is returned, which is defined as zero:

```
#define VSL_STATUS_OK 0
```

In case of an error, a non-zero error code is returned, which indicates the origin of the failure. The following status codes for the convolution/correlation error codes are pre-defined in the header files.

Convolution/Correlation Status Codes

Status Code	Description
<code>VSL_CC_ERROR_NOT_IMPLEMENTED</code>	Requested functionality is not implemented.
<code>VSL_CC_ERROR_ALLOCATION_FAILURE</code>	Memory allocation failure.
<code>VSL_CC_ERROR_BAD_DESCRIPTOR</code>	Task descriptor is corrupted.
<code>VSL_CC_ERROR_SERVICE_FAILURE</code>	A service function has failed.
<code>VSL_CC_ERROR_EDIT_FAILURE</code>	Failure while editing the task.
<code>VSL_CC_ERROR_EDIT_PROHIBITED</code>	You cannot edit this parameter.
<code>VSL_CC_ERROR_COMMIT_FAILURE</code>	Task commitment has failed.
<code>VSL_CC_ERROR_COPY_FAILURE</code>	Failure while copying the task.
<code>VSL_CC_ERROR_DELETE_FAILURE</code>	Failure while deleting the task.
<code>VSL_CC_ERROR_BAD_ARGUMENT</code>	Bad argument or task parameter.
<code>VSL_CC_ERROR_JOB</code>	Bad parameter: <i>job</i> .
<code>SL_CC_ERROR_KIND</code>	Bad parameter: <i>kind</i> .

Status Code	Description
VSL_CC_ERROR_MODE	Bad parameter: <i>mode</i> .
VSL_CC_ERROR_METHOD	Bad parameter: <i>method</i> .
VSL_CC_ERROR_TYPE	Bad parameter: <i>type</i> .
VSL_CC_ERROR_EXTERNAL_PRECISION	Bad parameter: <i>external_precision</i> .
VSL_CC_ERROR_INTERNAL_PRECISION	Bad parameter: <i>internal_precision</i> .
VSL_CC_ERROR_PRECISION	Incompatible external/internal precisions.
VSL_CC_ERROR_DIMS	Bad parameter: <i>dims</i> .
VSL_CC_ERROR_XSHAPE	Bad parameter: <i>xshape</i> .
VSL_CC_ERROR_YSHAPE	Bad parameter: <i>yshape</i> .
	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_CC_ERROR_ZSHAPE	Bad parameter: <i>zshape</i> .
VSL_CC_ERROR_XSTRIDE	Bad parameter: <i>xstride</i> .
VSL_CC_ERROR_YSTRIDE	Bad parameter: <i>ystride</i> .
VSL_CC_ERROR_ZSTRIDE	Bad parameter: <i>zstride</i> .
VSL_CC_ERROR_X	Bad parameter: <i>x</i> .
VSL_CC_ERROR_Y	Bad parameter: <i>y</i> .
VSL_CC_ERROR_Z	Bad parameter: <i>z</i> .
VSL_CC_ERROR_START	Bad parameter: <i>start</i> .
VSL_CC_ERROR_DECIMATION	Bad parameter: <i>decimation</i> .
VSL_CC_ERROR_OTHER	Another error.

Convolution and Correlation Task Constructors

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters. No additional parameter adjustment is typically required and other routines can use the task object.

Intel® MKL implementation of the convolution and correlation API provides two different forms of constructors: a general form and an X-form. X-form constructors work in the same way as the general form constructors but also assign particular data to the first operand vector used in the convolution or correlation operation (stored in array *x*).

Using X-form constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector held in array *x* against different vectors held in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Each constructor routine has an associated one-dimensional version that provides algorithmic and computational benefits.

NOTE

If the constructor fails to create a task descriptor, it returns the `NULL` task pointer.

The [Table "Task Constructors"](#) lists available task constructors:

Task Constructors

Routine	Description
vslConvNewTask/vslCorrNewTask	Creates a new convolution or correlation task descriptor for a multidimensional case.
vslConvNewTask1D/vslCorrNewTask1D	Creates a new convolution or correlation task descriptor for a one-dimensional case.
vslConvNewTaskX/vslCorrNewTaskX	Creates a new convolution or correlation task descriptor as an X-form for a multidimensional case.
vslConvNewTaskX1D/vslCorrNewTaskX1D	Creates a new convolution or correlation task descriptor as an X-form for a one-dimensional case.

[vslConvNewTask/vslCorrNewTask](#)

Creates a new convolution or correlation task descriptor for multidimensional case.

Syntax

```
status = vslsConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vsldConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslcConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslzConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslsCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vsldCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslcCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslzCorrNewTask(task, mode, dims, xshape, yshape, zshape);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>mode</code>	<code>const MKL_INT</code>	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
<code>dims</code>	<code>const MKL_INT</code>	Rank of user data. Specifies number of dimensions for the input and output arrays <i>x</i> , <i>y</i> , and <i>z</i> used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.

Name	Type	Description
<i>xshape</i>	<code>const int[]</code>	Defines the shape of the input data for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	<code>const int[]</code>	Defines the shape of the input data for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	<code>const int[]</code>	Defines the shape of the output data to be stored in array <i>z</i> . See Data Allocation for more information.

Output Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr* for vslsConvNewTask, vsldConvNewTask, vslcConvNewTask, vslzConvNewTask VSLCorrTaskPtr* for vslsCorrNewTask, vsldCorrNewTask, vslcConvNewTask, vslzConvNewTask	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
<i>status</i>	<code>int</code>	Set to <code>VSL_STATUS_OK</code> if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTask/vslCorrNewTask` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)).

The parameters *xshape*, *yshape*, and *zshape* define the shapes of the input and output data provided by the arrays *x*, *y*, and *z*, respectively. Each shape parameter is an array of integers with its length equal to the value of *dims*. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter *dims* is 1, then *xshape*, *yshape*, *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. Note that values of shape parameters may differ from physical shapes of arrays *x*, *y*, and *z* if non-trivial strides are assigned.

If the constructor fails to create a task descriptor, it returns a NULL task pointer.

vslConvNewTask1D/vslCorrNewTask1D

Creates a new convolution or correlation task descriptor for one-dimensional case.

Syntax

```
status = vslsConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vsldConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslcConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslzConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslsCorrNewTask1D(task, mode, xshape, yshape, zshape);
```

```
status = vsldCorrNewTask1D(task, mode, xshape, yshape, zshape);
status = vslcCorrNewTask1D(task, mode, xshape, yshape, zshape);
status = vslzCorrNewTask1D(task, mode, xshape, yshape, zshape);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>mode</i>	const MKL_INT	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
<i>xshape</i>	const MKL_INT	Defines the length of the input data sequence for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	const MKL_INT	Defines the length of the input data sequence for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	const MKL_INT	Defines the length of the output data sequence to be stored in array <i>z</i> . See Data Allocation for more information.

Output Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr* for vslsConvNewTask1D, vsldConvNewTask1D, vslcConvNewTask1D, vslzConvNewTask1D VSLCorrTaskPtr* for vslsCorrNewTask1D, vsldCorrNewTask1D, vslcCorrNewTask1D, vslzCorrNewTask1D	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
<i>status</i>	int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTask1D/vslCorrNewTask1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)). Unlike `vslConvNewTask/vslCorrNewTask`, these routines represent a special one-dimensional version of the constructor which assumes that the value of the parameter *dims* is 1. The parameters *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. You explicitly assign the shape parameters when calling the constructor.

vslConvNewTaskX/vslCorrNewTaskX

Creates a new convolution or correlation task descriptor for multidimensional case and assigns source data to the first operand vector.

Syntax

```
status = vslsConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vsldConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslcConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslzConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslsCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vsldCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslcCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslzCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>mode</i>	const MKL_INT	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
<i>dims</i>	const MKL_INT	Rank of user data. Specifies number of dimensions for the input and output arrays <i>x</i> , <i>y</i> , and <i>z</i> used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
<i>xshape</i>	const int[]	Defines the shape of the input data for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	const int[]	Defines the shape of the input data for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	const int[]	Defines the shape of the output data to be stored in array <i>z</i> . See Data Allocation for more information.
<i>x</i>	const float[] for real data in single precision flavors, const double[] for real data in double precision flavors, const MKL_Complex8[] for complex data in single precision flavors, const MKL_Complex16[] for complex data in double precision flavors	Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.

Name	Type	Description
<i>xstride</i>	const int[]	Strides for input data in the array <i>x</i> .

Output Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr* for vslsConvNewTaskX, vsldConvNewTaskX, vslcConvNewTaskX, vslzConvNewTaskX VSLCorrTaskPtr* for vslsCorrNewTaskX, vsldCorrNewTaskX, vslcCorrNewTaskX, vslzCorrNewTaskX	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
<i>status</i>	int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTaskX/vslCorrNewTaskX` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)).

Unlike `vslConvNewTask/vslCorrNewTask`, these routines represent the so called X-form version of the constructor, which means that in addition to creating the task descriptor they assign particular data to the first operand vector in array *x* used in convolution or correlation operation. The task descriptor created by the `vslConvNewTaskX/vslCorrNewTaskX` constructor keeps the pointer to the array *x* all the time, that is, until the task object is deleted by one of the destructor routines (see `vslConvDeleteTask/vslCorrDeleteTask`).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters *xshape*, *yshape*, and *zshape* define the shapes of the input and output data provided by the arrays *x*, *y*, and *z*, respectively. Each shape parameter is an array of integers with its length equal to the value of *dims*. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter *dims* is 1, then *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. Note that values of shape parameters may differ from physical shapes of arrays *x*, *y*, and *z* if non-trivial strides are assigned.

The stride parameter *xstride* specifies the physical location of the input data in the array *x*. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *xstride* is *s*, then only every *s*th element of the array *x* will be used to form the input sequence. The stride value must be positive or negative but not zero.

`vslConvNewTaskX1D/vslCorrNewTaskX1D`

Creates a new convolution or correlation task descriptor for one-dimensional case and assigns source data to the first operand vector.

Syntax

```

status = vs1sConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vs1dConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vs1cConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vs1zConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vs1sCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vs1dCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vs1cCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vs1zCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);

```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>mode</code>	<code>const MKL_INT</code>	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
<code>xshape</code>	<code>const MKL_INT</code>	Defines the length of the input data sequence for the source array <code>x</code> . See Data Allocation for more information.
<code>yshape</code>	<code>const MKL_INT</code>	Defines the length of the input data sequence for the source array <code>y</code> . See Data Allocation for more information.
<code>zshape</code>	<code>const MKL_INT</code>	Defines the length of the output data sequence to be stored in array <code>z</code> . See Data Allocation for more information.
<code>x</code>	<code>const float[]</code> for real data in single precision flavors, <code>const double[]</code> for real data in double precision flavors, <code>const MKL_Complex8[]</code> for complex data in single precision flavors, <code>const MKL_Complex16[]</code> for complex data in double precision flavors	Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.
<code>xstride</code>	<code>const MKL_INT</code>	Stride for input data sequence in the array <code>x</code> .

Output Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr* for vslsConvNewTaskX1D, vsldConvNewTaskX1D, vslcConvNewTaskX1D, vslzConvNewTaskX1D VSLCorrTaskPtr* for vslsCorrNewTaskX1D, vsldCorrNewTaskX1D, vslcCorrNewTaskX1D, vslzCorrNewTaskX1D	Pointer to the task descriptor if created successfully or NULL pointer otherwise.
<i>status</i>	int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTaskX1D/vslCorrNewTaskX1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)).

These routines represent a special one-dimensional version of the so called X-form of the constructor. This assumes that the value of the parameter *dims* is 1 and that in addition to creating the task descriptor, constructor routines assign particular data to the first operand vector in array *x* used in convolution or correlation operation. The task descriptor created by the `vslConvNewTaskX1D/vslCorrNewTaskX1D` constructor keeps the pointer to the array *x* all the time, that is, until the task object is deleted by one of the destructor routines (see `vslConvDeleteTask/vslCorrDeleteTask`).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters *xshape*, *yshape*, and *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. You explicitly assign the shape parameters when calling the constructor.

The [stride parameters](#) *xstride* specifies the physical location of the input data in the array *x* and is an interval between locations of consecutive elements of the array. For example, if the value of the parameter *xstride* is *s*, then only every *s*th element of the array *x* will be used to form the input sequence. The stride value must be positive or negative but not zero.

Convolution and Correlation Task Editors

Task editors in convolution and correlation API of Intel® oneAPI Math Kernel Library (oneMKL) are routines intended for setting up or changing the following task parameters (see [Table "Convolution and Correlation Task Parameters"](#)):

- *mode*
- *internal_precision*
- *start*
- *decimation*

For setting up or changing each of the above parameters, a separate routine exists.

NOTE

Fields of the task descriptor structure are accessible only through the set of task editor routines provided with the software.

The work data computed during the last commitment process may become invalid with respect to new parameter settings. That is why after applying any of the editor routines to change the task descriptor settings, the task loses its commitment status and goes through the full commitment process again during the next execution or copy operation. For more information on task commitment, see the [Introduction to Convolution and Correlation](#).

Table "Task Editors" lists available task editors.

Task Editors

Routine	Description
vslConvSetMode/vslCorrSetMode	Changes the value of the parameter <i>mode</i> for the operation of convolution or correlation.
vslConvSetInternalPrecision/vslCorrSetInternalPrecision	Changes the value of the parameter <i>internal_precision</i> for the operation of convolution or correlation.
vslConvSetStart/vslCorrSetStart	Sets the value of the parameter <i>start</i> for the operation of convolution or correlation.
vslConvSetDecimation/vslCorrSetDecimation	Sets the value of the parameter <i>decimation</i> for the operation of convolution or correlation.

NOTE

You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

[vslConvSetMode/vslCorrSetMode](#)

*Changes the value of the parameter *mode* in the convolution or correlation task descriptor.*

Syntax

```
status = vslConvSetMode(task, newmode);
```

```
status = vslCorrSetMode(task, newmode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr for vslConvSetMode VSLCorrTaskPtr for vslCorrSetMode	Pointer to the task descriptor.
<i>newmode</i>	const MKL_INT	New value of the parameter <i>mode</i> .

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task.

Description

This function is declared in `mkl_vsl_functions.h`.

The function routine changes the value of the parameter `mode` for the operation of convolution or correlation. This parameter defines whether the computation should be done via Fourier transforms of the input/output data or using a direct algorithm. Initial value for `mode` is assigned by a task constructor.

Predefined values for the `mode` parameter are as follows:

Values of *mode* parameter

Value	Purpose
VSL_CONV_MODE_FFT	Compute convolution by using fast Fourier transform.
VSL_CORR_MODE_FFT	Compute correlation by using fast Fourier transform.
VSL_CONV_MODE_DIRECT	Compute convolution directly.
VSL_CORR_MODE_DIRECT	Compute correlation directly.
VSL_CONV_MODE_AUTO	Automatically choose direct or Fourier mode for convolution.
VSL_CORR_MODE_AUTO	Automatically choose direct or Fourier mode for correlation.

vslConvSetInternalPrecision/vslCorrSetInternalPrecision

Changes the value of the parameter `internal_precision` in the convolution or correlation task descriptor.

Syntax

```
status = vslConvSetInternalPrecision(task, precision);
```

```
status = vslCorrSetInternalPrecision(task, precision);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr for vslConvSetInternalPrecision VSLCorrTaskPtr for vslCorrSetInternalPrecision	Pointer to the task descriptor.
<i>precision</i>	const MKL_INT	New value of the parameter <i>internal_precision</i> .

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task.

Description

The `vslConvSetInternalPrecision/vslCorrSetInternalPrecision` routine changes the value of the parameter *internal_precision* for the operation of convolution or correlation. This parameter defines whether the internal computations of the convolution or correlation result should be done in single or double precision. Initial value for *internal_precision* is assigned by a task constructor and set to either "single" or "double" according to the particular flavor of the constructor used.

Changing the *internal_precision* can be useful if the default setting of this parameter was "single" but you want to calculate the result with double precision even if input and output data are represented in single precision.

Predefined values for the *internal_precision* input parameter are as follows:

Values of *internal_precision* Parameter

Value	Purpose
VSL_CONV_PRECISION_SINGLE	Compute convolution with single precision.
VSL_CORR_PRECISION_SINGLE	Compute correlation with single precision.
VSL_CONV_PRECISION_DOUBLE	Compute convolution with double precision.
VSL_CORR_PRECISION_DOUBLE	Compute correlation with double precision.

`vslConvSetStart/vslCorrSetStart`

Changes the value of the parameter start in the convolution or correlation task descriptor.

Syntax

```
status = vslConvSetStart(task, start);
```

```
status = vslCorrSetStart(task, start);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr for <code>vslConvSetStart</code> VSLCorrTaskPtr for <code>vslCorrSetStart</code>	Pointer to the task descriptor.
<i>start</i>	const int[]	New value of the parameter <i>start</i> .

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task.

Description

The `vslConvSetStart/vslCorrSetStart` routine sets the value of the parameter *start* for the operation of convolution or correlation. In a one-dimensional case, this parameter points to the first element in the mathematical result that should be stored in the output array. In a multidimensional case, *start* is an array of indices and its length is equal to the number of dimensions specified by the parameter *dims*. For more information about the definition and effect of this parameter, see [Data Allocation](#).

During the initial task descriptor construction, the default value for *start* is undefined and this parameter is not used. Therefore the only way to set and use the *start* parameter is via assigning it some value by one of the `vslConvSetStart/vslCorrSetStart` routines.

vslConvSetDecimation/vslCorrSetDecimation

Changes the value of the parameter decimation in the convolution or correlation task descriptor.

Syntax

```
status = vslConvSetDecimation(task, decimation);
status = vslCorrSetDecimation(task, decimation);
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr for vslConvSetDecimation VSLCorrTaskPtr for vslCorrSetDecimation	Pointer to the task descriptor.
<i>decimation</i> <i>n</i>	const int[]	New value of the parameter <i>decimation</i> .

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task.

Description

The routine sets the value of the parameter *decimation* for the operation of convolution or correlation. This parameter determines how to thin out the mathematical result of convolution or correlation before writing it into the output data array. For example, in a one-dimensional case, if *decimation* = *d* > 1, only every *d*-th element of the mathematical result is written to the output array *z*. In a multidimensional case, *decimation* is an array of indices and its length is equal to the number of dimensions specified by the parameter *dims*. For more information about the definition and effect of this parameter, see [Data Allocation](#).

During the initial task descriptor construction, the default value for *decimation* is undefined and this parameter is not used. Therefore the only way to set and use the *decimation* parameter is via assigning it some value by one of the `vslSetDecimation` routines.

Task Execution Routines

Task execution routines compute convolution or correlation results based on parameters held by the task descriptor and on the user data supplied for input vectors.

After you create and adjust a task, you can execute it multiple times by applying to different input/output data of the same type, precision, and shape.

Intel® oneAPI Math Kernel Library (oneMKL) provides the following forms of convolution/correlation execution routines:

- **General form** executors that use the task descriptor created by the general form constructor and expect to get two source data arrays *x* and *y* on input
- **X-form** executors that use the task descriptor created by the X-form constructor and expect to get only one source data array *y* on input because the first array *x* has been already specified on the construction stage

When the task is executed for the first time, the execution routine includes a task commitment operation, which involves two basic steps: parameters consistency check and preparation of auxiliary data (for example, this might be the calculation of Fourier transform for input data).

Each execution routine has an associated one-dimensional version that provides algorithmic and computational benefits.

NOTE

You can use the `NULL` task pointer in calls to execution routines. In this case, the routine is terminated and no system crash occurs.

If the task is executed successfully, the execution routine returns the zero status code. If an error is detected, the execution routine returns an error code which signals that a specific error has occurred. In particular, an error status code is returned in the following cases:

- if the task pointer is `NULL`
- if the task descriptor is corrupted
- if calculation has failed for some other reason.

NOTE

Intel® MKL does not control floating-point errors, like overflow or gradual underflow, or operations with NaNs, etc.

If an error occurs, the task descriptor stores the error code.

The table below lists all task execution routines.

Task Execution Routines

Routine	Description
<code>vslConvExec/vslCorrExec</code>	Computes convolution or correlation for a multidimensional case.
<code>vslConvExec1D/vslCorrExec1D</code>	Computes convolution or correlation for a one-dimensional case.
<code>vslConvExecX/vslCorrExecX</code>	Computes convolution or correlation as X-form for a multidimensional case.
<code>vslConvExecX1D/vslCorrExecX1D</code>	Computes convolution or correlation as X-form for a one-dimensional case.

vslConvExec/vslCorrExec

Computes convolution or correlation for multidimensional case.

Syntax

```
status = vslsConvExec(task, x, xstride, y, ystride, z, zstride);
status = vsldConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslcConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslzConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslsCorrExec(task, x, xstride, y, ystride, z, zstride);
status = vsldCorrExec(task, x, xstride, y, ystride, z, zstride);
status = vslcCorrExec(task, x, xstride, y, ystride, z, zstride);
status = vslzCorrExec(task, x, xstride, y, ystride, z, zstride);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr for vslsConvExec, vsldConvExec, vslcConvExec, vslzConvExec VSLCorrTaskPtr for vslsCorrExec, vsldCorrExec, vslcCorrExec, vslzCorrExec	Pointer to the task descriptor
<i>x, y</i>	const float[] for vslsConvExec and vslsCorrExec, const double[] for vsldConvExec and vsldCorrExec, const MKL_Complex8[] for vslcConvExec and vslcCorrExec, const MKL_Complex16[] for vslzConvExec and vslzCorrExec	Pointers to arrays containing input data. See Data Allocation for more information.
<i>xstride, ystride, zstride</i>	const int[]	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<code>z</code>	const float[] for <code>vslsConvExec</code> and <code>vslsCorrExec</code> , const double[] for <code>vsldConvExec</code> and <code>vsldCorrExec</code> , const MKL_Complex8[] for <code>vslcConvExec</code> and <code>vslcCorrExec</code> , const MKL_Complex16[] for <code>vslzConvExec</code> and <code>vslzCorrExec</code>	Pointer to the array that stores output data. See Data Allocation for more information.
<code>status</code>	int	Set to <code>VSL_STATUS_OK</code> if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExec/vslCorrExec` routines computes convolution or correlation of the data provided by the arrays `x` and `y` and then stores the results in the array `z`. Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTask/vslCorrNewTask` constructor and pointed to by `task`. If `task` is `NULL`, no operation is done.

The stride parameters `xstride`, `ystride`, and `zstride` specify the physical location of the input and output data in the arrays `x`, `y`, and `z`, respectively. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter `zstride` is `s`, then only every s^{th} element of the array `z` will be used to store the output data. The stride value must be positive or negative but not zero.

vslConvExec1D/vslCorrExec1D

Computes convolution or correlation for one-dimensional case.

Syntax

```
status = vslsConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vsldConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslcConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslzConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslsCorrExec1D(task, x, xstride, y, ystride, z, zstride);
status = vsldCorrExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslcCorrExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslzCorrExec1D(task, x, xstride, y, ystride, z, zstride);
```

Include Files

- `mkh.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr for vs1sConvExec1D, vs1dConvExec1D, vs1cConvExec1D, vs1zConvExec1D VSLCorrTaskPtr for vs1sCorrExec1D, vs1dCorrExec1D, vs1cCorrExec1D, vs1zCorrExec1D	Pointer to the task descriptor.
<i>x, y</i>	const float[] for vs1sConvExec1D and vs1sCorrExec1D, const double[] for vs1dConvExec1D and vs1dCorrExec1D, const MKL_Complex8[] for vs1cConvExec1D and vs1cCorrExec1D, const MKL_Complex16[] for vs1zConvExec1D and vs1zCorrExec1D	Pointers to arrays containing input data. See Data Allocation for more information.
<i>xstride, ystride, zstride</i>	const MKL_INT	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<i>z</i>	const float[] for vs1sConvExec1D and vs1sCorrExec1D, const double[] for vs1dConvExec1D and vs1dCorrExec1D, const MKL_Complex8[] for vs1cConvExec1D and vs1cCorrExec1D, const MKL_Complex16[] for vs1zConvExec1D and vs1zCorrExec1D	Pointer to the array that stores output data. See Data Allocation for more information.

Name	Type	Description
<i>status</i>	int	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExec1D/vslCorrExec1D` routines computes convolution or correlation of the data provided by the arrays *x* and *y* and then stores the results in the array *z*. These routines represent a special one-dimensional version of the operation, assuming that the value of the parameter *dims* is 1. Using this version of execution routines can help speed up performance in case of one-dimensional data.

Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTask1D/vslCorrNewTask1D` constructor and pointed to by *task*. If *task* is NULL, no operation is done.

`vslConvExecX/vslCorrExecX`

Computes convolution or correlation for multidimensional case with the fixed first operand vector.

Syntax

```
status = vslsConvExecX(task, y, ystride, z, zstride);
status = vsldConvExecX(task, y, ystride, z, zstride);
status = vslcConvExecX(task, y, ystride, z, zstride);
status = vslzConvExecX(task, y, ystride, z, zstride);
status = vslsCorrExecX(task, y, ystride, z, zstride);
status = vslcCorrExecX(task, y, ystride, z, zstride);
status = vslzCorrExecX(task, y, ystride, z, zstride);
status = vsldCorrExecX(task, y, ystride, z, zstride);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr for vslsConvExecX, vsldConvExecX, vslcConvExecX, vslzConvExecX VSLCorrTaskPtr for vslsCorrExecX, vsldCorrExecX, vslcCorrExecX, vslzCorrExecX	Pointer to the task descriptor.

Name	Type	Description
<i>x</i> , <i>y</i>	const float[] for vslsConvExecX and vslsCorrExecX, const double[] for vsldConvExecX and vsldCorrExecX, const MKL_Complex8[] for vslcConvExecX and vslcCorrExecX, const MKL_Complex16[] for vslzConvExecX and vslzCorrExecX	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.
<i>ystride</i> , <i>z</i> , <i>stride</i>	const int[]	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<i>z</i>	const float[] for vslsConvExecX and vslsCorrExecX, const double[] for vsldConvExecX and vsldCorrExecX, const MKL_Complex8[] for vslcConvExecX and vslcCorrExecX, const MKL_Complex16[] for vslzConvExecX and vslzCorrExecX	Pointer to the array that stores output data. See Data Allocation for more information.
<i>status</i>	int	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExecX/vslCorrExecX` routines computes convolution or correlation of the data provided by the arrays *x* and *y* and then stores the results in the array *z*. These routines represent a special version of the operation, which assumes that the first operand vector was set on the task construction stage and the task object keeps the pointer to the array *x*.

Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTaskX/vslCorrNewTaskX` constructor and pointed to by *task*. If *task* is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple convolutions or correlations with the same data vector in array *x* against different vectors in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

vslConvExecX1D/vslCorrExecX1D

Computes convolution or correlation for one-dimensional case with the fixed first operand vector.

Syntax

```
status = vslsConvExecX1D(task, y, ystride, z, zstride);
status = vsldConvExecX1D(task, y, ystride, z, zstride);
status = vslcConvExecX1D(task, y, ystride, z, zstride);
status = vslzConvExecX1D(task, y, ystride, z, zstride);
status = vslsCorrExecX1D(task, y, ystride, z, zstride);
status = vsldCorrExecX1D(task, y, ystride, z, zstride);
status = vslcCorrExecX1D(task, y, ystride, z, zstride);
status = vslzCorrExecX1D(task, y, ystride, z, zstride);
status = vsldCorrExecX1D(task, y, ystride, z, zstride);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	VSLConvTaskPtr for vslsConvExecX1D, vsldConvExecX1D, vslcConvExecX1D, vslzConvExecX1D VSLCorrTaskPtr for vslsCorrExecX1D, vsldCorrExecX1D, vslcCorrExecX1D, vslzCorrExecX1D	Pointer to the task descriptor.
<i>x, y</i>	const float[] for vslsConvExecX1D and vslsCorrExecX1D, const double[] for vsldConvExecX1D and vsldCorrExecX1D, const MKL_Complex8[] for vslcConvExecX1D and vslcCorrExecX1D,	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.

Name	Type	Description
	<code>const MKL_Complex16[]</code> for <code>vslzConvExecX1D</code> and <code>vslzCorrExecX1D</code>	
<code>ystride,</code> <code>zstride</code>	<code>const MKL_INT</code>	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<code>z</code>	<code>const float[]</code> for <code>vslsConvExecX1D</code> and <code>vslsCorrExecX1D</code> , <code>const double[]</code> for <code>vsldConvExecX1D</code> and <code>vsldCorrExecX1D</code> , <code>const MKL_Complex8[]</code> for <code>vslcConvExecX1D</code> and <code>vslcCorrExecX1D</code> , <code>const MKL_Complex16[]</code> for <code>vslzConvExecX1D</code> and <code>vslzCorrExecX1D</code>	Pointer to the array that stores output data. See Data Allocation for more information.
<code>status</code>	<code>int</code>	Set to <code>VSL_STATUS_OK</code> if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExecX1D/vslCorrExecX1D` routines computes convolution or correlation of one-dimensional (assuming that `dims = 1`) data provided by the arrays `x` and `y` and then stores the results in the array `z`. These routines represent a special version of the operation, which expects that the first operand vector was set on the task construction stage.

Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTaskX1D/vslCorrNewTaskX1D` constructor and pointed to by `task`. If `task` is `NULL`, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple one-dimensional convolutions or correlations with the same data vector in array `x` against different vectors in array `y`. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Convolution and Correlation Task Destructors

Task destructors are routines designed for deleting task objects and deallocating memory.

`vslConvDeleteTask/vslCorrDeleteTask`

Destroys the task object and frees the memory.

Syntax

```
errcode = vslConvDeleteTask(task);
```

```
errcode = vslConvDeleteTask(task);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	VslConvTaskPtr* for vslConvDeleteTask VslCorrTaskPtr* for vslCorrDeleteTask	Pointer to the task descriptor.

Output Parameters

Name	Type	Description
<i>errcode</i>	int	Contains 0 if the task object is deleted successfully. Contains an error code if an error occurred.

Description

The `vslConvDeleteTask/vslCorrvDeleteTask` routine deletes the task descriptor object and frees any working memory and the memory allocated for the data structure. The task pointer is set to `NULL`.

Note that if the `vslConvDeleteTask/vslCorrvDeleteTask` routine does not delete the task successfully, the routine returns an error code. This error code has no relation to the task status code and does not change it.

NOTE

You can use the `NULL` task pointer in calls to destructor routines. In this case, the routine terminates with no system crash.

Convolution and Correlation Task Copiers

The routines are designed for copying convolution and correlation task descriptors.

`vslConvCopyTask/vslCorrCopyTask`

Copies a descriptor for convolution or correlation task.

Syntax

```
status = vslConvCopyTask(newtask, srctask);
status = vslCorrCopyTask(newtask, srctask);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>srctask</i>	const VSLConvTaskPtr for vslConvCopyTask const VSLCorrTaskPtr for vslCorrCopyTask	Pointer to the source task descriptor.

Output Parameters

Name	Type	Description
<i>newtask</i>	VSLConvTaskPtr* for vslConvCopyTask VSLCorrTaskPtr* for vslCorrCopyTask	Pointer to the new task descriptor.
<i>status</i>	int	Current status of the source task.

Description

If a task object *srctask* already exists, you can use an appropriate `vslConvCopyTask/vslCorrCopyTask` routine to make its copy in *newtask*. After the copy operation, both source and new task objects will become committed (see [Introduction to Convolution and Correlation](#) for information about task commitment). If the source task was not previously committed, the commitment operation for this task is implicitly invoked before copying starts. If an error occurs during source task commitment, the task stores the error code in the status field. If an error occurs during copy operation, the routine returns a `NULL` pointer instead of a reference to a new task object.

Convolution and Correlation Usage Examples

This section demonstrates how you can use the Intel® oneAPI Math Kernel Library (oneMKL) routines to perform some common convolution and correlation operations both for single-threaded and multithreaded calculations. The following two sample functions `scond1` and `sconf1` simulate the convolution and correlation functions `SCOND` and `SCONF` found in IBM ESSL* library. The functions assume single-threaded calculations and can be used with C or C++ compilers.

Function `scond1` for Single-Threaded Calculations

```
#include "mkl_vsl.h"

int acond1(
    float h[], int inch,
    float x[], int incx,
    float y[], int incy,
    int nh, int nx, int iy0, int ny)
{
    int status;
    VSLConvTaskPtr task;
    vslsConvNewTask1D(&task, VSL_CONV_MODE_DIRECT, nh, nx, ny);
    vslConvSetStart(task, &iy0);
    status = vslsConvExec1D(task, h, inch, x, incx, y, incy);
    vslConvDeleteTask(&task);
    return status;
}
```

Function `sconf1` for Single-Threaded Calculations

```
#include "mkl_vsl.h"

int sconf1(
    int init,
    float h[], int inclh,
    float x[], int inclx, int inc2x,
    float y[], int incly, int inc2y,
    int nh, int nx, int m, int iy0, int ny,
    void* aux1, int naux1, void* aux2, int naux2)
{
    int status;
    /* assume that aux1!=0 and naux1 is big enough */
    VSLConvTaskPtr* task = (VSLConvTaskPtr*)aux1;
    if (init != 0)
        /* initialization: */
        status = vslsConvNewTaskX1D(task, VSL_CONV_MODE_FFT,
            nh, nx, ny, h, inclh);
    if (init == 0) {
        /* calculations: */
        int i;
        vslConvSetStart(*task, &iy0);
        for (i=0; i<m; i++) {
            float* xi = &x[inc2x * i];
            float* yi = &y[inc2y * i];
            /* task is implicitly committed at i==0 */
            status = vslsConvExecX1D(*task, xi, inclx, yi, incly);
        };
    };
    vslConvDeleteTask(task);
    return status;
}
```

Using Multiple Threads

For functions such as `sconf1` described in the previous example, parallel calculations may be more preferable instead of cycling. If $m > 1$, you can use multiple threads for invoking the task execution against different data sequences. For such cases, use task copy routines to create m copies of the task object before the calculations stage and then run these copies with different threads. Ensure that you make all necessary parameter adjustments for the task (using [Task Editors](#)) before copying it.

The sample code in this case may look as follows:

```
if (init == 0) {
    int i, status, ss[M];
    VSLConvTaskPtr tasks[M];
    /* assume that M is big enough */
    . . .
    vslConvSetStart(*task, &iy0);
    . . .
    for (i=0; i<m; i++)
        /* implicit commitment at i==0 */
        vslConvCopyTask(&tasks[i], *task);
    . . .
```

Then, m threads may be started to execute different copies of the task:

```
. . .
    float* xi = &x[inc2x * i];
    float* yi = &y[inc2y * i];
    ss[i]=vslsConvExecX1D(tasks[i], xi, inclx, yi, incly);
    . . .
```

And finally, after all threads have finished the calculations, overall status should be collected from all task objects. The following code signals the first error found, if any:

```
. . .
    for (i=0; i<m; i++) {
        status = ss[i];
        if (status != 0) /* 0 means "OK" */
            break;
    };
    return status;
}; /* end if init==0 */
```

Execution routines modify the task internal state (fields of the task structure). Such modifications may conflict with each other if different threads work with the same task object simultaneously. That is why different threads must use different copies of the task.

Product and Performance Information

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

Notice revision #20201201

Convolution and Correlation Mathematical Notation and Definitions

The following notation is necessary to explain the underlying mathematical definitions used in the text:

$\mathbf{R} = (-\infty, +\infty)$	The set of real numbers.
$\mathbf{Z} = \{0, \pm 1, \pm 2, \dots\}$	The set of integer numbers.
$\mathbf{Z}^N = \mathbf{Z} \times \dots \times \mathbf{Z}$	The set of N-dimensional series of integer numbers.
$p = (p_1, \dots, p_N) \in \mathbf{Z}^N$	N-dimensional series of integers.
$u: \mathbf{Z}^N \rightarrow \mathbf{R}$	Function u with arguments from \mathbf{Z}^N and values from \mathbf{R} .
$u(p) = u(p_1, \dots, p_N)$	The value of the function u for the argument (p_1, \dots, p_N) .
$w = u * v$	Function w is the convolution of the functions u, v .
$w = u \bullet v$	Function w is the correlation of the functions u, v .

Given series $p, q \in \mathbf{Z}^N$:

- series $r = p + q$ is defined as $r^n = p^n + q^n$ for every $n=1, \dots, N$
- series $r = p - q$ is defined as $r^n = p^n - q^n$ for every $n=1, \dots, N$
- series $r = \sup\{p, q\}$ is defined as $r^n = \max\{p^n, q^n\}$ for every $n=1, \dots, N$
- series $r = \inf\{p, q\}$ is defined as $r^n = \min\{p^n, q^n\}$ for every $n=1, \dots, N$
- inequality $p \leq q$ means that $p^n \leq q^n$ for every $n=1, \dots, N$.

A function $u(p)$ is called a finite function if there exist series $p^{\min}, p^{\max} \in \mathbf{Z}^N$ such that:

$$u(p) \neq 0$$

implies

$$p^{\min} \leq p \leq p^{\max}.$$

Operations of convolution and correlation are only defined for finite functions.

Consider functions u, v and series $p^{\min}, p^{\max}, q^{\min}, q^{\max} \in \mathbf{Z}^N$ such that:

$$u(p) \neq 0 \text{ implies } p^{\min} \leq p \leq p^{\max}.$$

$$v(q) \neq 0 \text{ implies } q^{\min} \leq q \leq q^{\max}.$$

Definitions of linear correlation and linear convolution for functions u and v are given below.

Linear Convolution

If function $w = u * v$ is the convolution of u and v , then:

$$w(r) \neq 0 \text{ implies } \mathbf{R}^{\min} \leq r \leq \mathbf{R}^{\max},$$

$$\text{where } \mathbf{R}^{\min} = p^{\min} + q^{\min} \text{ and } \mathbf{R}^{\max} = p^{\max} + q^{\max}.$$

If $\mathbf{R}^{\min} \leq r \leq \mathbf{R}^{\max}$, then:

$$w(r) = \sum u(t) \cdot v(r-t) \text{ is the sum for all } t \in \mathbf{Z}^N \text{ such that } \mathbf{T}^{\min} \leq t \leq \mathbf{T}^{\max},$$

$$\text{where } \mathbf{T}^{\min} = \sup\{p^{\min}, r - q^{\max}\} \text{ and } \mathbf{T}^{\max} = \inf\{p^{\max}, r - q^{\min}\}.$$

Linear Correlation

If function $w = u \bullet v$ is the correlation of u and v , then:

$w(r) \neq 0$ implies $\mathbf{R}^{\min} \leq r \leq \mathbf{R}^{\max}$,
 where $\mathbf{R}^{\min} = \mathbf{Q}^{\min} - \mathbf{P}^{\max}$ and $\mathbf{R}^{\max} = \mathbf{Q}^{\max} - \mathbf{P}^{\min}$.

If $\mathbf{R}^{\min} \leq r \leq \mathbf{R}^{\max}$, then:

$w(r) = \sum u(t) \cdot v(r+t)$ is the sum for all $t \in \mathbf{Z}^N$ such that $\mathbf{T}^{\min} \leq t \leq \mathbf{T}^{\max}$,
 where $\mathbf{T}^{\min} = \sup\{\mathbf{P}^{\min}, \mathbf{Q}^{\min}-r\}$ and $\mathbf{T}^{\max} = \inf\{\mathbf{P}^{\max}, \mathbf{Q}^{\max}-r\}$.

Representation of the functions u , v , was the input/output data for the Intel® oneAPI Math Kernel Library (oneMKL) convolution and correlation functions is described in the [Data Allocation](#).

Convolution and Correlation Data Allocation

This section explains the relation between:

- mathematical finite functions u , v , w introduced in [Mathematical Notation and Definitions](#);
- multi-dimensional input and output data vectors representing the functions u , v , w ;
- arrays u , v , w used to store the input and output data vectors in computer memory

The convolution and correlation routine parameters that determine the allocation of input and output data are the following:

- Data arrays x , y , z
- Shape arrays $xshape$, $yshape$, $zshape$
- Strides within arrays $xstride$, $ystride$, $zstride$
- Parameters $start$, $decimation$

Finite Functions and Data Vectors

The finite functions $u(p)$, $v(q)$, and $w(r)$ introduced above are represented as multi-dimensional vectors of input and output data:

`inputu(i1, ..., idims)` for $u(p_1, \dots, p_N)$

`inputv(j1, ..., jdims)` for $v(q_1, \dots, q_N)$

`output(k1, ..., kdims)` for $w(r_1, \dots, r_N)$.

Parameter *dims* represents the number of dimensions and is equal to N.

The parameters *xshape*, *yshape*, and *zshape* define the shapes of input/output vectors:

`inputu(i1, ..., idims)` is defined if $1 \leq i_n \leq xshape(n)$ for every $n=1, \dots, dims$

`inputv(j1, ..., jdims)` is defined if $1 \leq j_n \leq yshape(n)$ for every $n=1, \dots, dims$

`output(k1, ..., kdims)` is defined if $1 \leq k_n \leq zshape(n)$ for every $n=1, \dots, dims$.

Relation between the input vectors and the functions u and v is defined by the following formulas:

`inputu(i1, ..., idims)` = $u(p_1, \dots, p_N)$, where $p_n = P_n^{\min} + (i_n - 1)$ for every n

`inputv(j1, ..., jdims)` = $v(q_1, \dots, q_N)$, where $q_n = Q_n^{\min} + (j_n - 1)$ for every n .

The relation between the output vector and the function $w(r)$ is similar (but only in the case when parameters *start* and *decimation* are not defined):

`output(k1, ..., kdims)` = $w(r_1, \dots, r_N)$, where $r_n = R_n^{\min} + (k_n - 1)$ for every n .

If the parameter *start* is defined, it must belong to the interval $R_n^{\min} \leq start(n) \leq R_n^{\max}$. If defined, the *start* parameter replaces R^{\min} in the formula:

`output(k1, ..., kdims)` = $w(r_1, \dots, r_N)$, where $r_n = start(n) + (k_n - 1)$

If the parameter *decimation* is defined, it changes the relation according to the following formula:

`output(k1, ..., kdims)` = $w(r_1, \dots, r_N)$, where $r_n = R_n^{\min} + (k_n - 1) * decimation(n)$

If both parameters *start* and *decimation* are defined, the formula is as follows:

$\text{output}(k_1, \dots, k_{\text{dims}}) = w(r_1, \dots, r_N)$, where $r_n = \text{start}(n) + (k_n - 1) \cdot \text{decimation}(n)$

The convolution and correlation software checks the values of *zshape*, *start*, and *decimation* during task commitment. If r_n exceeds R_n^{\max} for some $k_n, n=1, \dots, \text{dims}$, an error is raised.

Allocation of Data Vectors

Both parameter arrays *x* and *y* contain input data vectors in memory, while array *z* is intended for storing output data vector. To access the memory, the convolution and correlation software uses only pointers to these arrays and ignores the array shapes.

For parameters *x*, *y*, and *z*, you can provide one-dimensional arrays with the requirement that actual length of these arrays be sufficient to store the data vectors.

The allocation of the input and output data inside the arrays *x*, *y*, and *z* is described below assuming that the arrays are one-dimensional. Given multi-dimensional indices $i, j, k \in \mathbf{Z}^N$, one-dimensional indices $e, f, g \in \mathbf{Z}$ are defined such that:

$\text{inputu}(i_1, \dots, i_{\text{dims}})$ is allocated at $x(e)$

$\text{inputv}(j_1, \dots, j_{\text{dims}})$ is allocated at $y(f)$

$\text{output}(k_1, \dots, k_{\text{dims}})$ is allocated at $z(g)$.

The indices *e*, *f*, and *g* are defined as follows:

$e = 1 + \sum x_{\text{stride}}(n) \cdot dx(n)$ (the sum is for all $n=1, \dots, \text{dims}$)

$f = 1 + \sum y_{\text{stride}}(n) \cdot dy(n)$ (the sum is for all $n=1, \dots, \text{dims}$)

$g = 1 + \sum z_{\text{stride}}(n) \cdot dz(n)$ (the sum is for all $n=1, \dots, \text{dims}$)

The distances $dx(n)$, $dy(n)$, and $dz(n)$ depend on the signum of the stride:

$dx(n) = i_n - 1$ if $x_{\text{stride}}(n) > 0$, or $dx(n) = i_n - x_{\text{shape}}(n)$ if $x_{\text{stride}}(n) < 0$

$dy(n) = j_n - 1$ if $y_{\text{stride}}(n) > 0$, or $dy(n) = j_n - y_{\text{shape}}(n)$ if $y_{\text{stride}}(n) < 0$

$dz(n) = k_n - 1$ if $z_{\text{stride}}(n) > 0$, or $dz(n) = k_n - z_{\text{shape}}(n)$ if $z_{\text{stride}}(n) < 0$

The definitions of indices *e*, *f*, and *g* assume that indexes for arrays *x*, *y*, and *z* are started from unity:

$x(e)$ is defined for $e=1, \dots, \text{length}(x)$

$y(f)$ is defined for $f=1, \dots, \text{length}(y)$

$z(g)$ is defined for $g=1, \dots, \text{length}(z)$

Below is a detailed explanation about how elements of the multi-dimensional output vector are stored in the array *z* for one-dimensional and two-dimensional cases.

One-dimensional case. If $\text{dims}=1$, then *zshape* is the number of the output values to be stored in the array *z*. The actual length of array *z* may be greater than *zshape* elements.

If $z_{\text{stride}} > 1$, output values are stored with the stride: $\text{output}(1)$ is stored to $z(1)$, $\text{output}(2)$ is stored to $z(1+z_{\text{stride}})$, and so on. Hence, the actual length of *z* must be at least $1+z_{\text{stride}} \cdot (z_{\text{shape}}-1)$ elements or more.

If $z_{\text{stride}} < 0$, it still defines the stride between elements of array *z*. However, the order of the used elements is the opposite. For the *k*-th output value, $\text{output}(k)$ is stored in $z(1+|z_{\text{stride}}| \cdot (z_{\text{shape}}-k))$, where $|z_{\text{stride}}|$ is the absolute value of z_{stride} . The actual length of the array *z* must be at least $1+|z_{\text{stride}}| \cdot (z_{\text{shape}} - 1)$ elements.

Two-dimensional case. If $dims=2$, the output data is a two-dimensional matrix. The value `zstride(1)` defines the stride inside matrix columns, that is, the stride between the `output(k1, k2)` and `output(k1+1, k2)` for every pair of indices k_1, k_2 . On the other hand, `zstride(2)` defines the stride between columns, that is, the stride between `output(k1, k2)` and `output(k1, k2+1)`.

If `zstride(2)` is greater than `zshape(1)`, this causes sparse allocation of columns. If the value of `zstride(2)` is smaller than `zshape(1)`, this may result in the transposition of the output matrix. For example, if `zshape = (2, 3)`, you can define `zstride = (3, 1)` to allocate output values like transposed matrix of the shape 3×2 .

Whether `zstride` assumes this kind of transformations or not, you need to ensure that different elements output (k_1, \dots, k_{dims}) will be stored in different locations $z(g)$.

Summary Statistics

The Summary Statistics domain provides routines that compute basic statistical estimates for single and double precision multi-dimensional datasets.

The Summary Statistics routines calculate:

- raw and central moments up to the fourth order
- skewness and excess kurtosis (further referred to as *kurtosis* for brevity)
- variation coefficient
- quantiles and order statistics
- minimum and maximum
- variance-covariance/correlation matrix
- pooled/group variance-covariance matrix and mean
- partial variance-covariance/correlation matrix
- robust estimators for variance-covariance matrix and mean in presence of outliers
- raw/central partial sums up to the fourth order (for brevity referred to as *raw/central sums*)
- matrix of cross-products and sums of squares (for brevity referred to as *cross-product matrix*)
- median absolute deviation, mean absolute deviation

The library also contains functions to perform the following tasks:

- Detect outliers in datasets
- Support missing values in datasets
- Parameterize correlation matrices
- Compute quantiles for streaming data

[Mathematical Notation and Definitions](#) defines the supported operations in the Summary Statistics routines.

You can access the Summary Statistics routines through the Fortran 90 and C89 language interfaces. You can use the C89 interface with later versions of the C/C++.

The `mkl_vsl.h` header file is in the `${MKL}/include` directory.

You can find examples that demonstrate calculation of the Summary Statistics estimates in the `${MKL}/examples/vslc` example directory.

The Summary Statistics API is implemented through task objects, or tasks. A task object is a data structure, or a descriptor, holding parameters that determine a specific Summary Statistics operation. For example, such parameters may be precision, dimensions of user data, the matrix of the observations, or shapes of data arrays.

All the Summary Statistics routines process a task object as follows:

1. Create a task.
2. Modify settings of the task parameters.
3. Compute statistical estimates.
4. Destroy the task.

The Summary Statistics functions fall into the following categories:

Task Constructors - routines that create a new task object descriptor and set up most common parameters (dimension, number of observations, and matrix of the observations).

Task Editors - routines that can set or modify some parameter settings in the existing task descriptor.

Task Computation Routine - a routine that computes specified statistical estimates.

Task Destructor - a routine that deletes the task object and frees the memory.

A Summary Statistics task object contains a series of pointers to the input and output data arrays. You can read and modify the datasets and estimates at any time but you should allocate and release memory for such data.

See detailed information on the algorithms, API, and their usage in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* [SS Notes].

Summary Statistics Naming Conventions

The names of Summary Statistics routines, types, and constants are case-sensitive and can contain lowercase and uppercase characters (`vslsSSEditQuantiles`).

The names of routines have the following structure:

```
vsl[datatype]SS<base name>
```

where

- `vsl` is a prefix indicating that the routine belongs to Intel® oneAPI Math Kernel Library (oneMKL) Vector Statistics.
- `[datatype]` specifies the type of the input and/or output data and can be `s` (single precision real type), `d` (double precision real type), or `i` (integer type).
- `SS/ss` indicates that the routine is intended for calculations of the Summary Statistics estimates.
- `<base name>` specifies a particular functionality that the routine is designed for, for example, `NewTask`, `Compute`, `DeleteTask`.

NOTE

The Summary Statistics routine `vslDeleteTask` for deletion of the task is independent of the data type and its name omits the `[datatype]` field.

On 64-bit platforms, routines with the `_64` suffix support large data arrays in the LP64 interface library and enable you to mix integer types in one application. For more interface library details, see "Using the ILP64 Interface vs. LP64 Interface" in the developer guide.

Summary Statistics Data Types

The Summary Statistics routines use the following data types for calculations:

Type	Data Object
<code>VSLSSTaskPtr</code>	Pointer to a Summary Statistics task
<code>float</code>	Input/output user data in single precision
<code>double</code>	Input/output user data in double precision
<code>MKL_INT</code> or <code>long long</code>	Other data

NOTE

The actual size of the generic integer type is platform-specific and can be 32 or 64 bits in length. Before you compile your application, set an appropriate size for integers. See details in the 'Using the ILP64 Interface vs. LP64 Interface' section of the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

Summary Statistics Parameters

The basic parameters in the task descriptor (addresses of dimensions, number of observations, and datasets) are assigned values when the task editors create or modify the task object. Other parameters are determined by the specific task and changed by the task editors.

Summary Statistics Task Status and Error Reporting

The task status is an integer value, which is zero if no error is detected, or a specific non-zero error code otherwise. Negative status values indicate errors, and positive values indicate warnings. An error can be caused by invalid parameter values or a memory allocation failure.

The header files define symbolic names for the status codes. These names are defined as macros via `#define` statements.

The header files define the following status codes for the Summary Statistics error codes:

Summary Statistics Status Codes

Status Code	Description
VSL_STATUS_OK	Operation is successfully completed.
VSL_SS_ERROR_ALLOCATION_FAILURE	Memory allocation has failed.
VSL_SS_ERROR_BAD_DIMEN	Dimension value is invalid.
VSL_SS_ERROR_BAD_OBSERV_N	Invalid number (zero or negative) of observations was obtained.
VSL_SS_ERROR_STORAGE_NOT_SUPPORTED	Storage format is not supported.
VSL_SS_ERROR_BAD_INDC_ADDR	Array of indices is not defined.
VSL_SS_ERROR_BAD_WEIGHTS	Array of weights contains negative values.
VSL_SS_ERROR_BAD_MEAN_ADDR	Array of means is not defined.
VSL_SS_ERROR_BAD_2R_MOM_ADDR	Array of the second order raw moments is not defined.
VSL_SS_ERROR_BAD_3R_MOM_ADDR	Array of the third order raw moments is not defined.
VSL_SS_ERROR_BAD_4R_MOM_ADDR	Array of the fourth order raw moments is not defined.
VSL_SS_ERROR_BAD_2C_MOM_ADDR	Array of the second order central moments is not defined.
VSL_SS_ERROR_BAD_3C_MOM_ADDR	Array of the third order central moments is not defined.
VSL_SS_ERROR_BAD_4C_MOM_ADDR	Array of the fourth order central moments is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_KURTOSIS_ADDR	Array of kurtosis values is not defined.
VSL_SS_ERROR_BAD_SKEWNESS_ADDR	Array of skewness values is not defined.
VSL_SS_ERROR_BAD_MIN_ADDR	Array of minimum values is not defined.
VSL_SS_ERROR_BAD_MAX_ADDR	Array of maximum values is not defined.
VSL_SS_ERROR_BAD_VARIATION_ADDR	Array of variation coefficients is not defined.
VSL_SS_ERROR_BAD_COV_ADDR	Covariance matrix is not defined.
VSL_SS_ERROR_BAD_COR_ADDR	Correlation matrix is not defined.
VSL_SS_ERROR_BAD_QUANT_ORDER_ADDR	Array of quantile orders is not defined.
VSL_SS_ERROR_BAD_QUANT_ORDER	Quantile order value is invalid.
VSL_SS_ERROR_BAD_QUANT_ADDR	Array of quantiles is not defined.
VSL_SS_ERROR_BAD_ORDER_STATS_ADDR	Array of order statistics is not defined.
VSL_SS_ERROR_MOMORDER_NOT_SUPPORTED	Moment of requested order is not supported.
VSL_SS_NOT_FULL_RANK_MATRIX	Correlation matrix is not of full rank.
VSL_SS_ERROR_ALL_OBSERVS_OUTLIERS	All observations are outliers. (At least one observation must not be an outlier.)
VSL_SS_ERROR_BAD_ROBUST_COV_ADDR	Robust covariance matrix is not defined.
VSL_SS_ERROR_BAD_ROBUST_MEAN_ADDR	Array of robust means is not defined.
VSL_SS_ERROR_METHOD_NOT_SUPPORTED	Requested method is not supported.
VSL_SS_ERROR_NULL_TASK_DESCRIPTOR	Task descriptor is null.
VSL_SS_ERROR_BAD_OBSERV_ADDR	Dataset matrix is not defined.
VSL_SS_ERROR_BAD_ACCUM_WEIGHT_ADDR	Pointer to the variable that holds the value of accumulated weight is not defined.
VSL_SS_ERROR_SINGULAR_COV	Covariance matrix is singular.
VSL_SS_ERROR_BAD_POOLED_COV_ADDR	Pooled covariance matrix is not defined.
VSL_SS_ERROR_BAD_POOLED_MEAN_ADDR	Array of pooled means is not defined.
VSL_SS_ERROR_BAD_GROUP_COV_ADDR	Group covariance matrix is not defined.
VSL_SS_ERROR_BAD_GROUP_MEAN_ADDR	Array of group means is not defined.
VSL_SS_ERROR_BAD_GROUP_INDC_ADDR	Array of group indices is not defined.
VSL_SS_ERROR_BAD_GROUP_INDC	Group indices have improper values.
VSL_SS_ERROR_BAD_OUTLIERS_PARAMS_ADDR	Array of parameters for the outlier detection algorithm is not defined.
VSL_SS_ERROR_BAD_OUTLIERS_PARAMS_N_ADDR	Pointer to size of the parameter array for the outlier detection algorithm is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_OUTLIERS_WEIGHTS_ADDR	Output of the outlier detection algorithm is not defined.
VSL_SS_ERROR_BAD_ROBUST_COV_PARAMS_ADDR	Array of parameters of the robust covariance estimation algorithm is not defined.
VSL_SS_ERROR_BAD_ROBUST_COV_PARAMS_N_ADDR	Pointer to the number of parameters of the algorithm for robust covariance is not defined.
VSL_SS_ERROR_BAD_STORAGE_ADDR	Pointer to the variable that holds the storage format is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COV_IDX_ADDR	Array that encodes sub-components of a random vector for the partial covariance algorithm is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COV_IDX	Array that encodes sub-components of a random vector for partial covariance has improper values.
VSL_SS_ERROR_BAD_PARTIAL_COV_ADDR	Partial covariance matrix is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COR_ADDR	Partial correlation matrix is not defined.
VSL_SS_ERROR_BAD_MI_PARAMS_ADDR	Array of parameters for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_PARAMS_N_ADDR	Pointer to number of parameters for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_BAD_PARAMS_N	Size of the parameter array of the Multiple Imputation method is invalid.
VSL_SS_ERROR_BAD_MI_PARAMS	Parameters of the Multiple Imputation method are invalid.
VSL_SS_ERROR_BAD_MI_INIT_ESTIMATES_N_ADDR	Pointer to the number of initial estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_INIT_ESTIMATES_ADDR	Array of initial estimates for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_ADDR	Array of simulated missing values in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_N_ADDR	Pointer to the size of the array of simulated missing values in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_ESTIMATES_N_ADDR	Pointer to the number of parameter estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_ESTIMATES_ADDR	Array of parameter estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_N	Invalid size of the array of simulated values in the Multiple Imputation method.
VSL_SS_ERROR_BAD_MI_ESTIMATES_N	Invalid size of an array to hold parameter estimates obtained using the Multiple Imputation method.

Status Code	Description
VSL_SS_ERROR_BAD_MI_OUTPUT_PARAMS	Array of output parameters in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_PRIOR_N_ADDR	Pointer to the number of prior parameters is not defined.
VSL_SS_ERROR_BAD_MI_PRIOR_ADDR	Array of prior parameters is not defined.
VSL_SS_ERROR_BAD_MI_MISSING_VALS_N	Invalid number of missing values was obtained.
VSL_SS_SEMIDEFINITE_COR	Correlation matrix passed into the parameterization function is semi-definite.
VSL_SS_ERROR_BAD_PARAMTR_COR_ADDR	Correlation matrix to be parameterized is not defined.
VSL_SS_ERROR_BAD_COR	All eigenvalues of the correlation matrix to be parameterized are non-positive.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_N_ADDR	Pointer to the number of parameters for the quantile computation algorithm for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_ADDR	Array of parameters of the quantile computation algorithm for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_N	Invalid number of parameters of the quantile computation algorithm for streaming data has been obtained.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS	Invalid parameters of the quantile computation algorithm for streaming data have been passed.
VSL_SS_ERROR_BAD_STREAM_QUANT_ORDER_ADDR	Array of the quantile orders for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_ORDER	Invalid quantile order for streaming data is defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_ADDR	Array of quantiles for streaming data is not defined.
VSL_SS_ERROR_BAD_SUM_ADDR	Array of sums is not defined.
VSL_SS_ERROR_BAD_2R_SUM_ADDR	Array of raw sums of 2nd order is not defined.
VSL_SS_ERROR_BAD_3R_SUM_ADDR	Array of raw sums of 3rd order is not defined.
VSL_SS_ERROR_BAD_4R_SUM_ADDR	Array of raw sums of 4th order is not defined.
VSL_SS_ERROR_BAD_2C_SUM_ADDR	Array of central sums of 2nd order is not defined.
VSL_SS_ERROR_BAD_3C_SUM_ADDR	Array of central sums of 3rd order is not defined.
VSL_SS_ERROR_BAD_4C_SUM_ADDR	Array of central sums of 4th order is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_CP_SUM_ADDR	Cross-product matrix is not defined.
VSL_SS_ERROR_BAD_MDAD_ADDR	Array of median absolute deviations is not defined.
VSL_SS_ERROR_BAD_MNAD_ADDR	Array of mean absolute deviations is not defined.
VSL_SS_ERROR_BAD_SORTED_OBSERV_ADDR	Array for storing observation sorting results is not defined.
VSL_SS_ERROR_ERROR_INDICES_NOT_SUPPORTED	Array of indices is not supported.

Routines for robust covariance estimation, outlier detection, partial covariance estimation, multiple imputation, and parameterization of a correlation matrix can return internal error codes that are related to a specific implementation. Such error codes indicate invalid input data or other bugs in the Intel® oneAPI Math Kernel Library (oneMKL) routines other than the Summary Statistics routines.

Summary Statistics Task Constructors

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters.

NOTE

If the constructor fails to create a task descriptor, it returns the `NULL` task pointer.

vsSSNewTask

Creates and initializes a new summary statistics task descriptor.

Syntax

```
status = vsSSNewTask(&task, p, n, xstorage, x, w, indices);
status = vsdSSNewTask(&task, p, n, xstorage, x, w, indices);
```

Include Files

- `mkh.h`

Input Parameters

Name	Type	Description
<i>p</i>	<code>const MKL_INT*</code>	Dimension of the task, number of variables
<i>n</i>	<code>const MKL_INT*</code>	Number of observations
<i>xstorage</i>	<code>const MKL_INT*</code>	Storage format of matrix of observations
<i>x</i>	<code>const float*</code> for <code>vsSSNewTask</code> <code>const double*</code> for <code>vsdSSNewTask</code>	Matrix of observations

Name	Type	Description
<i>w</i>	<code>const float*</code> for <code>vslsSSNewTask</code> <code>const double*</code> for <code>vsldSSNewTask</code>	Array of weights of size <i>n</i> . Elements of the arrays are non-negative numbers. If a <code>NULL</code> pointer is passed, each observation is assigned weight equal to 1.
<i>indices</i>	<code>const MKL_INT*</code>	Array of vector components that will be processed. Size of array is <i>p</i> . If a <code>NULL</code> pointer is passed, all components of random vector are processed.

Output Parameters

Name	Type	Description
<i>task</i>	<code>VSLSTaskPtr*</code>	Descriptor of the task
<i>status</i>	<code>int</code>	Set to <code>VSL_STATUS_OK</code> if the task is created successfully, otherwise a non-zero error code is returned.

Description

Each `vslsSSNewTask` constructor routine creates a new summary statistics task descriptor with the user-specified value for a required parameter, dimension of the task. The optional parameters (matrix of observations, its storage format, number of observations, weights of observations, and indices of the random vector components) are set to their default values.

The observations of random *p*-dimensional vector $\xi = (\xi_1, \dots, \xi_i, \dots, \xi_p)$, which are *n* vectors of dimension *p*, are passed as a one-dimensional array *x*. The parameter *xstorage* defines the storage format of the observations and takes one of the possible values listed in [Table "Storage format of matrix of observations and order statistics"](#).

Storage format of matrix of observations, order statistics, and matrix of sorted observations

Parameter	Description
<code>VSL_SS_MATRIX_STORAGE_ROWS</code>	The observations of random vector ξ are packed by rows: <i>n</i> data points for the vector component ξ_1 come first, <i>n</i> data points for the vector component ξ_2 come second, and so forth.
<code>VSL_SS_MATRIX_STORAGE_COLS</code>	The observations of random vector ξ are packed by columns: the first <i>p</i> -dimensional observation of the vector ξ comes first, the second <i>p</i> -dimensional observation of the vector comes second, and so forth.

A one-dimensional array *w* of size *n* contains non-negative weights assigned to the observations. You can pass a `NULL` array into the constructor. In this case, each observation is assigned the default value of the weight.

You can choose vector components for which you wish to compute statistical estimates. If an element of the vector *indices* of size *p* contains 0, the observations that correspond to this component are excluded from the calculations. If you pass the `NULL` value of the parameter into the constructor, statistical estimates for all random variables are computed.

If the constructor fails to create a task descriptor, it returns the `NULL` task pointer.

Summary Statistics Task Editors

Task editors are intended to set up or change the task parameters listed in [Table "Parameters of Summary Statistics Task to Be Initialized or Modified"](#). As an example, to compute the sample mean for a one-dimensional dataset, initialize a variable for the mean value, and pass its address into the task as shown in the example below:

```
#define DIM    1
#define N     1000

int main()
{
    VSLSSTaskPtr task;
    double x[N];
    double mean;
    MKL_INT p, n, xstorage;
    int status;
    /* initialize variables used in the computations of sample mean */
    p = DIM;
    n = N;
    xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
    mean = 0.0;

    /* create task */
    status = vslSSNewTask( &task, &p, &n, &xstorage, x, 0, 0 );

    /* initialize task parameters */
    status = vslSSEditTask( task, VSL_SS_ED_MEAN, &mean );

    /* compute mean using SS fast method */
    status = vslSSCompute(task, VSL_SS_MEAN, VSL_SS_METHOD_FAST );

    /* deallocate task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

Use the single (`vslssedittask`) or double (`vslDSSedittask`) version of an editor, to initialize single or double precision version task parameters, respectively. Use an integer version of an editor (`vslISSedittask`) to initialize parameters of the integer type.

[Table "Summary Statistics Task Editors"](#) lists the task editors for Summary Statistics. Each of them initializes and/or modifies a respective group of related parameters.

Summary Statistics Task Editors

Editor	Description
<code>vslSSeditTask</code>	Changes a pointer in the task descriptor.
<code>vslSSeditMoments</code>	Changes pointers to arrays associated with raw and central moments.
<code>vslSSeditSums</code>	Modifies the pointers to arrays that hold sum estimates.
<code>vslSSeditCovCor</code>	Changes pointers to arrays associated with covariance and/or correlation matrices.
<code>vslSSeditCP</code>	Modifies the pointers to cross-product matrix parameters.

Editor	Description
<code>vslSSEditPartialCovCor</code>	Changes pointers to arrays associated with partial covariance and/or correlation matrices.
<code>vslSSEditQuantiles</code>	Changes pointers to arrays associated with quantile/order statistics calculations.
<code>vslSSEditStreamQuantiles</code>	Changes pointers to arrays for quantile related calculations for streaming data.
<code>vslSSEditPooledCovariance</code>	Changes pointers to arrays associated with algorithms related to a pooled covariance matrix.
<code>vslSSEditRobustCovariance</code>	Changes pointers to arrays for robust estimation of a covariance matrix and mean.
<code>vslSSEditOutliersDetection</code>	Changes pointers to arrays for detection of outliers.
<code>vslSSEditMissingValues</code>	Changes pointers to arrays associated with the method of supporting missing values in a dataset.
<code>vslSSEditCorParameterization</code>	Changes pointers to arrays associated with the algorithm for parameterization of a correlation matrix.

NOTE

You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

vslSSEditTask

Modifies address of an input/output parameter in the task descriptor.

Syntax

```
status = vslsSSEditTask(task, parameter, par_addr);
status = vsldSSEditTask(task, parameter, par_addr);
status = vsliSSEditTask(task, parameter, par_addr);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>VSLSTaskPtr</code>	Descriptor of the task
<code>parameter</code>	<code>const MKL_INT</code>	Parameter to change
<code>par_addr</code>	<code>const float*</code> for <code>vslsSSEditTask</code> <code>const double*</code> for <code>vsldSSEditTask</code> <code>const MKL_INT*</code> for <code>vsliSSEditTask</code>	Address of the new parameter

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vslSSEditTask` routine replaces the pointer to the parameter stored in the Summary Statistics task descriptor with the `par_addr` pointer. If you pass the `NULL` pointer to the editor, no changes take place in the task and a corresponding error code is returned. See [Table "Parameters of Summary Statistics Task to Be Initialized or Modified"](#) for the predefined values of the parameter.

Use the single (`vslssedittask`) or double (`vsldssedittask`) version of the editor, to initialize single or double precision version task parameters, respectively. Use an integer version of the editor (`vslisedittask`) to initialize parameters of the integer type.

Parameters of Summary Statistics Task to Be Initialized or Modified

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_DIMEN	i	Address of a variable that holds the task dimension	Required. Positive integer value.
VSL_SS_ED_OBSERV_N	i	Address of a variable that holds the number of observations	Required. Positive integer value.
VSL_SS_ED_OBSERV	d, s	Address of the observation matrix	Required. Provide the matrix containing your observations.
VSL_SS_ED_OBSERV_STORAGE	i	Address of a variable that holds the storage format for the observation matrix	Required. Provide a storage format supported by the library whenever you pass a matrix of observations. ¹
VSL_SS_ED_INDC	i	Address of the array of indices	Optional. Provide this array if you need to process individual components of the random vector. Set entry <i>i</i> of the array to one to include the <i>i</i> th coordinate in the analysis. Set entry <i>i</i> of the array to zero to exclude the <i>i</i> th coordinate from the analysis.
VSL_SS_ED_WEIGHTS	d, s	Address of the array of observation weights	Optional. If the observations have weights different from the default weight (one), set entries of the array to non-negative floating point values.
VSL_SS_ED_MEAN	d, s	Address of the array of means	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_2R_MOM	d, s	Address of an array of raw moments of the second order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3R_MOM	d, s	Address of an array of raw moments of the third order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4R_MOM	d, s	Address of an array of raw moments of the fourth order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2C_MOM	d, s	Address of an array of central moments of the second order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first and second order.
VSL_SS_ED_3C_MOM	d, s	Address of an array of central moments of the third order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, and third order.
VSL_SS_ED_4C_MOM	d, s	Address of an array of central moments of the fourth order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, third, and fourth order.
VSL_SS_ED_KURTOSIS	d, s	Address of the array of kurtosis estimates	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, third, and fourth order.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_SKEWNESS	d, s	Address of the array of skewness estimates	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, and third order.
VSL_SS_ED_MIN	d, s	Address of the array of minimum estimates	Optional. Set entries of array to meaningful values, such as the values of the first observation.
VSL_SS_ED_MAX	d, s	Address of the array of maximum estimates	Optional. Set entries of array to meaningful values, such as the values of the first observation.
VSL_SS_ED_VARIATION	d, s	Address of the array of variation coefficients	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first and second order.
VSL_SS_ED_COV	d, s	Address of a covariance matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Make sure you also provide an array for the mean.
VSL_SS_ED_COV_STORAGE	i	Address of the variable that holds the storage format for a covariance matrix	Required. Provide a storage format supported by the library whenever you intend to compute the covariance matrix. ²
VSL_SS_ED_COR	d, s	Address of a correlation matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. If you initialize the matrix in non-trivial way, make sure that the main diagonal contains variance values. Also, provide an array for the mean.
VSL_SS_ED_COR_STORAGE	i	Address of the variable that holds the correlation storage format for a correlation matrix	Required. Provide a storage format supported by the library whenever you intend to compute the correlation matrix. ²

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_ACCUM_WEIGHT	d, s	Address of the array of size 2 that holds the accumulated weight (sum of weights) in the first position and the sum of weights squared in the second position	Optional. Set the entries of the matrix to meaningful values (typically zero) if you intend to do progressive processing of the dataset or need the sum of weights and sum of squared weights assigned to observations.
VSL_SS_ED_QUANT_ORDER_N	i	Address of the variable that holds the number of quantile orders	Required. Positive integer value. Provide the number of quantile orders whenever you compute quantiles.
VSL_SS_ED_QUANT_ORDER	d, s	Address of the array of quantile orders	Required. Set entries of array to values from the interval (0,1). Provide this parameter whenever you compute quantiles.
VSL_SS_ED_QUANT_QUANTILE_S	d, s	Address of the array of quantiles	None.
VSL_SS_ED_ORDER_STATS	d, s	Address of the array of order statistics	None.
VSL_SS_ED_GROUP_INDC	i	Address of the array of group indices used in computation of a pooled covariance matrix	Required. Set entry i to integer value k if the observation belongs to group k . Values of k take values in the range $[0, g-1]$, where g is the number of groups.
VSL_SS_ED_POOLED_COV_STORAGE	i	Address of a variable that holds the storage format for a pooled covariance matrix	Required. Provide a storage format supported by the library whenever you intend to compute pooled covariance. ²
VSL_SS_ED_POOLED_MEAN	d, s	Address of an array of pooled means	None.
VSL_SS_ED_POOLED_COV	d, s	Address of pooled covariance matrices	None.
VSL_SS_ED_GROUP_COV_INDC	i	Address of an array of indices for which covariance/means should be computed	Optional. Set the k th entry of the array to 1 if you need group covariance and mean for group k ; otherwise set it to zero.
VSL_SS_ED_REQ_GROUP_INDC	i	Address of an array of indices for which group estimates such as covariance or means are requested	Optional. Set the k th entry of the array to 1 if you need an estimate for group k ; otherwise set it to zero.
VSL_SS_ED_GROUP_MEANS	i	Address of an array of group means	None.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_GROUP_COV_STORAGE	d, s	Address of a variable that holds the storage format for a group covariance matrix	Required. Provide a storage format supported by the library whenever you intend to get group covariance. ²
VSL_SS_ED_GROUP_COV	d, s	Address of group covariance matrices	None.
VSL_SS_ED_ROBUST_COV_STORAGE	d, s	Address of a variable that holds the storage format for a robust covariance matrix	Required. Provide a storage format supported by the library whenever you compute robust covariance ² .
VSL_SS_ED_ROBUST_COV_PARAMS_N	i	Address of a variable that holds the number of algorithmic parameters of the method for robust covariance estimation	Required. Set to the number of TBS parameters, <i>VSL_SS_TBS_PARAMS_N</i> .
VSL_SS_ED_ROBUST_COV_PARAMS	d, s	Address of an array of parameters of the method for robust estimation of a covariance	Required. Set the entries of the array according to the description in vslSSEditRobustCovariance .
VSL_SS_ED_ROBUST_MEAN	i	Address of an array of robust means	None.
VSL_SS_ED_ROBUST_COV	d, s	Address of a robust covariance matrix	None.
VSL_SS_ED_OUTLIERS_PARAMS_N	d, s	Address of a variable that holds the number of parameters of the outlier detection method	Required. Set to the number of outlier detection parameters, <i>VSL_SS_BACON_PARAMS_N</i> .
VSL_SS_ED_OUTLIERS_PARAMS	i	Address of an array of algorithmic parameters for the outlier detection method	Required. Set the entries of the array according to the description in vslSSEditOutliersDetection .
VSL_SS_ED_OUTLIERS_WEIGHTS	d, s	Address of an array of weights assigned to observations by the outlier detection method	None.
VSL_SS_ED_ORDER_STATS_STORAGE	d, s	Address of a variable that holds the storage format of an order statistics matrix	Required. Provide a storage format supported by the library whenever you compute a matrix of order statistics. ¹
VSL_SS_ED_PARTIAL_COV_IDS	i	Address of an array that encodes subcomponents of a random vector	Required. Set the entries of the array according to the description in vslSSEditPartialCovCor .
VSL_SS_ED_PARTIAL_COV	d, s	Address of a partial covariance matrix	None.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_PARTIAL_COV_STORAGE	i	Address of a variable that holds the storage format of a partial covariance matrix	Required. Provide a storage format supported by the library whenever you compute the partial covariance. ²
VSL_SS_ED_PARTIAL_COR	d, s	Address of a partial correlation matrix	None.
VSL_SS_ED_PARTIAL_COR_STORAGE	i	Address of a variable that holds the storage format for a partial correlation matrix	Required. Provide a storage format supported by the library whenever you compute the partial correlation. ²
VSL_SS_ED_MI_PARAMS_N	i	Address of a variable that holds the number of algorithmic parameters for the Multiple Imputation method	Required. Set to the number of MI parameters, <code>VSL_SS_MI_PARAMS_SIZE</code> .
VSL_SS_ED_MI_PARAMS	d, s	Address of an array of algorithmic parameters for the Multiple Imputation method	Required. Set entries of the array according to the description in vslSSEditMissingValues .
VSL_SS_ED_MI_INIT_ESTIMATES_N	i	Address of a variable that holds the number of initial estimates for the Multiple Imputation method	Optional. Set to $p+p*(p+1)/2$, where p is the task dimension.
VSL_SS_ED_MI_INIT_ESTIMATES	d, s	Address of an array of initial estimates for the Multiple Imputation method	Optional. Set the values of the array according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics" in the <i>Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes</i> document [SS Notes].
VSL_SS_ED_MI_SIMUL_VALS_N	i	Address of a variable that holds the number of simulated values in the Multiple Imputation method	Optional. Positive integer indicating the number of missing points in the observation matrix.
VSL_SS_ED_MI_SIMUL_VALS	d, s	Address of an array of simulated values in the Multiple Imputation method	None.
VSL_SS_ED_MI_ESTIMATES_N	i	Address of a variable that holds the number of estimates obtained as a result of the Multiple Imputation method	Optional. Positive integer number defined according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics" in the <i>Intel® oneAPI Math Kernel</i>

Parameter Value	Type	Purpose	Initialization
			<i>Library (oneMKL) Summary Statistics Application Notes</i> document [SS Notes].
VSL_SS_ED_MI_ESTIMATES	d, s	Address of an array of estimates obtained as a result of the Multiple Imputation method	None.
VSL_SS_ED_MI_PRIOR_N	i	Address of a variable that holds the number of prior parameters for the Multiple Imputation method	Optional. If you pass a user-defined array of prior parameters, set this parameter to $(p^2+3*p+4)/2$, where p is the task dimension.
VSL_SS_ED_MI_PRIOR	d, s	Address of an array of prior parameters for the Multiple Imputation method	Optional. Set entries of the array of prior parameters according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics" in the <i>Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes</i> document [SS Notes].
VSL_SS_ED_PARAMTR_COR	d, s	Address of a parameterized correlation matrix	None.
VSL_SS_ED_PARAMTR_COR_STORAGE	i	Address of a variable that holds the storage format of a parameterized correlation matrix	Required. Provide a storage format supported by the library whenever you compute the parameterized correlation matrix. ²
VSL_SS_ED_STREAM_QUANT_PARAMS_N	i	Address of a variable that holds the number of parameters of a quantile computation method for streaming data	Required. Set to the number of quantile computation parameters, <i>VSL_SS_SQUANTS_ZW_PARAMS_N</i> .
VSL_SS_ED_STREAM_QUANT_PARAMS	d, s	Address of an array of parameters of a quantile computation method for streaming data	Required. Set the entries of the array according to the description in "Computing Quantiles for Streaming Data" in the <i>Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes</i> document [SS Notes].
VSL_SS_ED_STREAM_QUANT_ORDER_N	i	Address of a variable that holds the number of quantile orders for streaming data	Required. Positive integer value.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_STREAM_QUANT_ORDER	d, s	Address of an array of quantile orders for streaming data	Required. Set entries of the array to values from the interval (0,1). Provide this parameter whenever you compute quantiles.
VSL_SS_ED_STREAM_QUANT_QUANTILES	d, s	Address of an array of quantiles for streaming data	None.
VSL_SS_ED_SUM	d, s	Address of array of sums	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2R_SUM	d, s	Address of array of raw sums of 2nd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3R_SUM	d, s	Address of array of raw sums of 3rd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4R_SUM	d, s	Address of array of raw sums of 4th order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2C_SUM	d, s	Address of array of central sums of 2nd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3C_SUM	d, s	Address of array of central sums of 3rd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4C_SUM	d, s	Address of array of central sums of 4th order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_CP	d, s	Address of cross-product matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_MDAD	d, s	Address of array of median absolute deviations	None.
VSL_SS_ED_MNAD	d, s	Address of array of mean absolute deviations	None.
VSL_SS_ED_SORTED_OBSERV	d, s	Address of the array that stores sorted results	None.
VSL_SS_ED_SORTED_OBSERV_STORAGE	i	Address of a variable that holds the storage format of an output matrix	Required. Provide a supported storage format whenever you specify sorting of the observation matrix.

1. See [Table: "Storage format of matrix of observations and order statistics"](#) for storage formats.
2. See [Table: "Storage formats of a variance-covariance/correlation matrix"](#) for storage formats.

vsLSSEditMoments

Modifies the pointers to arrays that hold moment estimates.

Syntax

```
status = vsLSSEditMoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m);
status = vsLdSEditMoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	VSLSSTaskPtr	Descriptor of the task
<i>mean</i>	float* for vsLSSEditMoments double* for vsLdSEditMoments	Pointer to the array of means
<i>r2m</i>	float* for vsLSSEditMoments double* for vsLdSEditMoments	Pointer to the array of raw moments of the 2 nd order
<i>r3m</i>	float* for vsLSSEditMoments double* for vsLdSEditMoments	Pointer to the array of raw moments of the 3 rd order
<i>r4m</i>	float* for vsLSSEditMoments double* for vsLdSEditMoments	Pointer to the array of raw moments of the 4 th order

Name	Type	Description
<i>c2m</i>	float* for <code>vsIsSSEditMoments</code> double* for <code>vsldSSEditMoments</code>	Pointer to the array of central moments of the 2 nd order
<i>c3m</i>	float* for <code>vsIsSSEditMoments</code> double* for <code>vsldSSEditMoments</code>	Pointer to the array of central moments of the 3 rd order
<i>c4m</i>	float* for <code>vsIsSSEditMoments</code> double* for <code>vsldSSEditMoments</code>	Pointer to the array of central moments of the 4 th order

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vsIsSSEditMoments` routine replaces pointers to the arrays that hold estimates of raw and central moments with values passed as corresponding parameters of the routine. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vsIsSSEditSums

Modifies the pointers to arrays that hold sum estimates.

Syntax

```
status = vsIsSSEditSums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s);
```

```
status = vsldSSEditSums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLSTaskPtr	Descriptor of the task
<i>sum</i>	float* for <code>vsIsSSEditSums</code> double* for <code>vsldSSEditSums</code>	Pointer to the array of sums
<i>r2s</i>	float* for <code>vsIsSSEditSums</code> double* for <code>vsldSSEditSums</code>	Pointer to the array of raw sums of the second order
<i>r3s</i>	float* for <code>vsIsSSEditSums</code> double* for <code>vsldSSEditSums</code>	Pointer to the array of raw sums of the third order
<i>r4s</i>	float* for <code>vsIsSSEditSums</code> double* for <code>vsldSSEditSums</code>	Pointer to the array of raw sums of the fourth order

Name	Type	Description
<i>c2s</i>	float* for vslsSSEditSums double* for vsldSSEditSums	Pointer to the array of central sums of the second order
<i>c3s</i>	float* for vslsSSEditSums double* for vsldSSEditSums	Pointer to the array of central sums of the third order
<i>c4s</i>	float* for vslsSSEditSums double* for vsldSSEditSums	Pointer to the array of central sums of the fourth order

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vsSSEditSums` routine replaces pointers to the arrays that hold estimates of raw and central sums with values passed as corresponding parameters of the routine. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vsSSEditCovCor

Modifies the pointers to covariance/correlation/cross-product parameters.

Syntax

```
status = vsSSEditCovCor(task, mean, cov, cov_storage, cor, cor_storage);
```

```
status = vsldSSEditCovCor(task, mean, cov, cov_storage, cor, cor_storage);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLSTaskPtr	Descriptor of the task
<i>mean</i>	float* for vsSSEditCovCor double* for vsldSSEditCovCor	Pointer to the array of means
<i>cov</i>	float* for vsSSEditCovCor double* for vsldSSEditCovCor	Pointer to a covariance matrix
<i>cov_storage</i>	const MKL_INT*	Pointer to the storage format of the covariance matrix
<i>cor</i>	float* for vsSSEditCovCor double* for vsldSSEditCovCor	Pointer to a correlation matrix

Name	Type	Description
<code>cor_storage</code>	<code>const MKL_INT*</code>	Pointer to the storage format of the correlation matrix

Output Parameters

Name	Type	Description
<code>status</code>	<code>int</code>	Current status of the task

Description

The `vslSSEditCovCor` routine replaces pointers to the array of means, covariance/correlation arrays, and their storage format with values passed as corresponding parameters of the routine. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

The storage parameters, `cov_storage` and `cor_storage`, describe the storage format used for the p -by- p symmetric variance-covariance/correlation/cross-product matrix C . The matrix C can be described as

$$C = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & \cdots & \cdots & c_{1,p} \\ c_{2,1} & c_{2,2} & \cdots & \cdots & \cdots & c_{2,p} \\ \vdots & \vdots & \ddots & & & \vdots \\ \vdots & \vdots & & c_{i,j} & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ c_{p,1} & c_{p,2} & \cdots & \cdots & \cdots & c_{p,p} \end{pmatrix}$$

Table "Storage formats of a variance-covariance/correlation/cross-product matrix" shows how the matrix is stored in a one-dimensional array cp for different values of the storage parameters.

Storage formats of variance-covariance/correlation/cross-product matrices

Parameter	Description
<code>VSL_SS_MATRIX_STORAGE_FULL</code>	<p>The array cp contains all elements of the matrix stored sequentially, row-by-row:</p> <ul style="list-style-type: none"> $cp[0]$ contains $c_{1,1}$ $cp[1]$ contains $c_{1,2}$ $cp[p-1]$ contains $c_{1,p}$ $cp[p]$ contains $c_{2,1}$ $cp[p*p-1]$ contains $c_{p,p}$ <p>The size of array cp is $p*p$.</p>
<code>VSL_SS_MATRIX_STORAGE_L_PACKED</code>	<p>The array cp contains the lower triangular part of the symmetric matrix stored sequentially, row-by-row:</p> <ul style="list-style-type: none"> $cp[0]$ contains $c_{1,1}$ $cp[1]$ contains $c_{2,1}$ $cp[2]$ contains $c_{2,2}$ and so on. <p>The size of the array is $p*(p+1)/2$.</p>
<code>VSL_SS_MATRIX_STORAGE_U_PACKED</code>	<p>The array cp contains the upper triangular part of the symmetric matrix stored sequentially, row-by-row:</p>

Parameter	Description
	$cp[0]$ contains $c_{1,1}$ $cp[1]$ contains $c_{1,2}$ $cp[3]$ contains $c_{1,3}$ and so on. The size of the array is $p*(p+1)/2$.

vsLSSEditCP

Modifies the pointers to cross-product matrix parameters.

Syntax

```
status = vsLSSEditCP(task, mean, sum, cp, cp_storage);
status = vsldSSEditCP(task, mean, sum, cp, cp_storage);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	VSLSTaskPtr	Descriptor of the task
<i>mean</i>	float* for vsLSSEditCP double* for vsldSSEditCP	Pointer to array of means
<i>sum</i>	float* for vsLSSEditCP double* for vsldSSEditCP	Pointer to array of sums
<i>cp</i>	float* for vsLSSEditCP double* for vsldSSEditCP	Pointer to a cross-product matrix
<i>cp_storage</i>	const MKL_INT*	Pointer to the storage format of the cross-product matrix

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The vsLSSEditCP routine replaces pointers to the array of means, array of sums, cross-product matrix, and its storage format with values passed as corresponding parameters of the routine. See [Table: "Storage formats of a variance-covariance/correlation/cross-product matrix"](#) for possible values of the *cp_storage* parameter. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Storage formats of variance-covariance/correlation/cross-product matrices

Parameter	Description
VSL_SS_MATRIX_STORAGE_FULL	<p>The array cp contains all elements of the matrix stored sequentially, row-by-row:</p> <p>$cp[0]$ contains $c_{1,1}$ $cp[1]$ contains $c_{1,2}$ $cp[p-1]$ contains $c_{1,p}$ $cp[p]$ contains $c_{2,1}$ $cp[p*p-1]$ contains $c_{p,p}$</p> <p>The size of array cp is $p*p$.</p>
VSL_SS_MATRIX_STORAGE_L_PACKED	<p>The array cp contains the lower triangular part of the symmetric matrix stored sequentially, row-by-row:</p> <p>$cp[0]$ contains $c_{1,1}$ $cp[1]$ contains $c_{2,1}$ $cp[2]$ contains $c_{2,2}$ and so on.</p> <p>The size of the array is $p*(p+1)/2$.</p>
VSL_SS_MATRIX_STORAGE_U_PACKED	<p>The array cp contains the upper triangular part of the symmetric matrix stored sequentially, row-by-row:</p> <p>$cp[0]$ contains $c_{1,1}$ $cp[1]$ contains $c_{1,2}$ $cp[3]$ contains $c_{1,3}$ and so on.</p> <p>The size of the array is $p*(p+1)/2$.</p>

vsLSSEditPartialCovCor

Modifies the pointers to partial covariance/correlation parameters.

Syntax

```
status = vsLSSEditPartialCovCor(task, p_idx_array, cov, cov_storage, cor, cor_storage,
p_cov, p_cov_storage, p_cor, p_cor_storage);
```

```
status = vsLdSSEditPartialCovCor(task, p_idx_array, cov, cov_storage, cor, cor_storage,
p_cov, p_cov_storage, p_cor, p_cor_storage);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	VSLSSTaskPtr	Descriptor of the task

Name	Type	Description
<i>p_idx_array</i>	const MKL_INT*	Pointer to the array that encodes indices of subcomponents Z and Y of the random vector as described in section Mathematical Notation and Definitions . <i>p_idx_array[i]</i> equals to -1 if the <i>i</i> -th component of the random vector belongs to Z 1, if the <i>i</i> -th component of the random vector belongs to Y.
<i>cov</i>	const float* for vslsSSEditPartialCovCor const double* for vsldSSEditPartialCovCor	Pointer to a covariance matrix
<i>cov_storage</i>	const MKL_INT*	Pointer to the storage format of the covariance matrix
<i>cor</i>	const float* for vslsSSEditPartialCovCor const double* for vsldSSEditPartialCovCor	Pointer to a correlation matrix
<i>cor_storage</i>	const MKL_INT*	Pointer to the storage format of the correlation matrix
<i>p_cov</i>	float* for vslsSSEditPartialCovCor double* for vsldSSEditPartialCovCor	Pointer to a partial covariance matrix
<i>p_cov_storage</i>	const MKL_INT*	Pointer to the storage format of the partial covariance matrix
<i>p_cor</i>	float* for vslsSSEditPartialCovCor double* for vsldSSEditPartialCovCor	Pointer to a partial correlation matrix
<i>p_cor_storage</i>	const MKL_INT*	Pointer to the storage format of the partial correlation matrix

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vslsSSEditPartialCovCor` routine replaces pointers to covariance/correlation arrays, partial covariance/correlation arrays, and their storage format with values passed as corresponding parameters of the routine. See [Table "Storage formats of a variance-covariance/correlation matrix"](#) for possible values of the *cov_storage*, *cor_storage*, *p_cov_storage*, and *p_cor_storage* parameters. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vsLSSEditQuantiles

Modifies the pointers to parameters related to quantile computations.

Syntax

```
status = vsLSSEditQuantiles(task, quant_order_n, quant_order, quants, order_stats,
order_stats_storage);
```

```
status = vsLdSSEditQuantiles(task, quant_order_n, quant_order, quants, order_stats,
order_stats_storage);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	VSLSTaskPtr	Descriptor of the task
<i>quant_order_n</i>	const MKL_INT*	Pointer to the number of quantile orders
<i>quant_order</i>	const float* for vsLSSEditQuantiles const double* for vsLdSSEditQuantiles	Pointer to the array of quantile orders
<i>quants</i>	float* for vsLSSEditQuantiles double* for vsLdSSEditQuantiles	Pointer to the array of quantiles
<i>order_stats</i>	float* for vsLSSEditQuantiles double* for vsLdSSEditQuantiles	Pointer to the array of order statistics
<i>order_stats_storage</i>	const MKL_INT*	Pointer to the storage format of the order statistics array

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vsLSSEditQuantiles` routine replaces pointers to the number of quantile orders, the array of quantile orders, the array of quantiles, the array that holds order statistics, and the storage format for the order statistics with values passed into the routine. See [Table "Storage format of matrix of observations and order statistics"](#) for possible values of the `order_statistics_storage` parameter. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vsLSSEditStreamQuantiles

Modifies the pointers to parameters related to quantile computations for streaming data.

Syntax

```
status = vslsSSEditStreamQuantiles(task, quant_order_n, quant_order, quants, nparams,
params);
```

```
status = vsldSSEditStreamQuantiles(task, quant_order_n, quant_order, quants, nparams,
params);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	VSLSSTaskPtr	Descriptor of the task
<i>quant_order_n</i>	const MKL_INT*	Pointer to the number of quantile orders
<i>quant_order</i>	const float* for vslsSSEditStreamQuantiles const double* for vsldSSEditStreamQuantiles	Pointer to the array of quantile orders
<i>quants</i>	float* for vslsSSEditStreamQuantiles double* for vsldSSEditStreamQuantiles	Pointer to the array of quantiles
<i>nparams</i>	const MKL_INT*	Pointer to the number of the algorithm parameters
<i>params</i>	const float* for vslsSSEditStreamQuantiles const double* for vsldSSEditStreamQuantiles	Pointer to the array of the algorithm parameters

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vslsSSEditStreamQuantiles` routine replaces pointers to the number of quantile orders, the array of quantile orders, the array of quantiles, the number of the algorithm parameters, and the array of the algorithm parameters with values passed into the routine. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vslSSEditPooledCovariance

Modifies pooled/group covariance matrix array pointers.

Syntax

```
status = vslsSSEditPooledCovariance(task, grp_indices, pld_mean, pld_cov,
req_grp_indices, grp_means, grp_cov);
```

```
status = vsldSSEditPooledCovariance(task, grp_indices, pld_mean, pld_cov,
req_grp_indices, grp_means, grp_cov);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	VSLSTaskPtr	Descriptor of the task
<i>grp_indices</i>	const MKL_INT*	Pointer to an array of size n . The i -th element of the array contains the number of the group the observation belongs to.
<i>pld_mean</i>	float* for vslsSSEditPooledCovariance double* for vsldSSEditPooledCovariance	Pointer to the array of pooled means
<i>pld_cov</i>	float* for vslsSSEditPooledCovariance double* for vsldSSEditPooledCovariance	Pointer to the array that holds a pooled covariance matrix
<i>req_grp_indices</i>	const MKL_INT*	Pointer to the array that contains indices of groups for which estimates to return (such as covariance and mean)
<i>grp_means</i>	float* for vslsSSEditPooledCovariance double* for vsldSSEditPooledCovariance	Pointer to the array of group means
<i>grp_cov</i>	float* for vslsSSEditPooledCovariance double* for vsldSSEditPooledCovariance	Pointer to the array that holds group covariance matrices

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vsLSSEditPooledCovariance` routine replaces pointers to the array of group indices, the array of pooled means, the array for a pooled covariance matrix, and pointers to the array of indices of group matrices, the array of group means, and the array for group covariance matrices with values passed in the editors. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.. Use the `vsLSSEditTask` routine to replace the storage format for pooled and group covariance matrices.

vsLSSEditRobustCovariance

Modifies pointers to arrays related to a robust covariance matrix.

Syntax

```
status = vsLSSEditRobustCovariance(task, rcov_storage, nparams, params, rmean, rcov);
status = vsldSSEditRobustCovariance(task, rcov_storage, nparams, params, rmean, rcov);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>VSLSTaskPtr</code>	Descriptor of the task
<code>rcov_storage</code>	<code>const MKL_INT*</code>	Pointer to the storage format of a robust covariance matrix
<code>nparams</code>	<code>const MKL_INT*</code>	Pointer to the number of method parameters
<code>params</code>	<code>const float*</code> for <code>vsLSSEditRobustCovariance</code> <code>const double*</code> for <code>vsldSSEditRobustCovariance</code>	Pointer to the array of method parameters
<code>rmean</code>	<code>float*</code> for <code>vsLSSEditRobustCovariance</code> <code>double*</code> for <code>vsldSSEditRobustCovariance</code>	Pointer to the array of robust means
<code>rcov</code>	<code>float*</code> for <code>vsLSSEditRobustCovariance</code> <code>double*</code> for <code>vsldSSEditRobustCovariance</code>	Pointer to a robust covariance matrix

Output Parameters

Name	Type	Description
<code>status</code>	<code>int</code>	Current status of the task

Description

The `vsLSSEditRobustCovariance` routine uses values passed as parameters of the routine to replace:

- pointers to covariance matrix storage
- pointers to the number of method parameters and to the array of the method parameters of size `nparams`
- pointers to the arrays that hold robust means and covariance

See [Table "Storage formats of a variance-covariance/correlation matrix"](#) for possible values of the `rcov_storage` parameter. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Intel® oneAPI Math Kernel Library (oneMKL) provides a Translated Biweight S-estimator (TBS) for robust estimation of a variance-covariance matrix and mean [Rocke96]. Use one iteration of the Maronna algorithm with the reweighting step [Maronna02] to compute the initial point of the algorithm. Pack the parameters of the TBS algorithm into the `params` array and pass them into the editor. [Table "Structure of the Array of TBS Parameters"](#) describes the `params` structure.

Structure of the Array of TBS Parameters

Array Position	Algorithm Parameter	Description
0	ϵ	Breakdown point, the number of outliers the algorithm can hold. By default, the value is $(n-p) / (2n)$.
1	α	Asymptotic rejection probability, see details in [Rocke96]. By default, the value is 0.001.
2	δ	Stopping criterion: the algorithm is terminated if weights are changed less than δ . By default, the value is 0.001.
3	<code>max_iter</code>	Maximum number of iterations. The algorithm terminates after <code>max_iter</code> iterations. By default, the value is 10. If you set this parameter to zero, the function returns a robust estimate of the variance-covariance matrix computed using the Maronna method [Maronna02] only.

The robust estimator of variance-covariance implementation in Intel® oneAPI Math Kernel Library (oneMKL) requires that the number of observations n be greater than twice the number of variables: $n > 2p$.

See additional details of the algorithm usage model in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [SS Notes].

vsLSSEditOutliersDetection

Modifies array pointers related to multivariate outliers detection.

Syntax

```
status = vsLSSEditOutliersDetection(task, nparams, params, w);
```

```
status = vsLDSEditOutliersDetection(task, nparams, params, w);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLSTaskPtr	Descriptor of the task
<i>nparams</i>	const MKL_INT*	Pointer to the number of method parameters
<i>params</i>	const float* for vslsSSEditOutliersDetection const double* for vsldSSEditOutliersDetection	Pointer to the array of method parameters
<i>w</i>	float* for vslsSSEditOutliersDetection double* for vsldSSEditOutliersDetection	Pointer to an array of size <i>n</i> . The array holds the weights of observations to be marked as outliers.

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vslsSSEditOutliersDetection` routine uses the parameters passed to replace

- the pointers to the number of method parameters and to the array of the method parameters of size *nparams*
- the pointer to the array that holds the calculated weights of the observations

If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Intel® oneAPI Math Kernel Library (oneMKL) provides the BACON algorithm ([Billor00]) for the detection of multivariate outliers. Pack the parameters of the BACON algorithm into the *params* array and pass them into the editor. [Table "Structure of the Array of BACON Parameters"](#) describes the *params* structure.

Structure of the Array of BACON Parameters

Array Position	Algorithm Parameter	Description
0	Method to start the algorithm	<p>The parameter takes one of the following possible values:</p> <p><code>VSL_SS_METHOD_BACON_MEDIAN_INIT</code>, if the algorithm is started using the median estimate. This is the default value of the parameter.</p> <p><code>VSL_SS_METHOD_BACON_MAHALANOBIS_INIT</code>, if the algorithm is started using the Mahalanobis distances.</p>

Array Position	Algorithm Parameter	Description
1	α	One-tailed probability that defines the $(1 - \alpha)$ quantile of χ^2 distribution with p degrees of freedom. The recommended value is α / n , where n is the number of observations. By default, the value is 0.05.
2	δ	Stopping criterion; the algorithm is terminated if the size of the basic subset is changed less than δ . By default, the value is 0.005.

Output of the algorithm is the vector of weights, `BaconWeights`, such that `BaconWeights(i) = 0` if i -th observation is detected as an outlier. Otherwise `BaconWeights(i) = w(i)`, where w is the vector of input weights. If you do not provide the vector of input weights, `BaconWeights(i)` is set to 1 if the i -th observation is not detected as an outlier.

See additional details about usage model of the algorithm in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [[SS Notes](#)].

vsLSSEditMissingValues

Modifies pointers to arrays associated with the method of supporting missing values in a dataset.

Syntax

```
status = vsLSSEditMissingValues(task, nparams, params, init_estimates_n,
init_estimates, prior_n, prior, simul_missing_vals_n, simul_missing_vals, estimates_n,
estimates);
```

```
status = vsLdSSEditMissingValues(task, nparams, params, init_estimates_n,
init_estimates, prior_n, prior, simul_missing_vals_n, simul_missing_vals, estimates_n,
estimates);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>VSLSSTaskPtr</code>	Descriptor of the task
<code>nparams</code>	<code>const MKL_INT*</code>	Pointer to the number of method parameters
<code>params</code>	<code>const float*</code> for <code>vsLSSEditMissingValues</code> <code>const double*</code> for <code>vsLdSSEditMissingValues</code>	Pointer to the array of method parameters
<code>init_estimates_n</code>	<code>const MKL_INT*</code>	Pointer to the number of initial estimates for mean and a variance-covariance matrix

Name	Type	Description
<i>init_estimates</i>	const float* for vslsSSEditMissingValues const double* for vsldsSSEditMissingValues	Pointer to the array that holds initial estimates for mean and a variance-covariance matrix
<i>prior_n</i>	const MKL_INT*	Pointer to the number of prior parameters
<i>prior</i>	const float* for vslsSSEditMissingValues const double* for vsldsSSEditMissingValues	Pointer to the array of prior parameters
<i>simul_missing_vals_n</i>	const MKL_INT*	Pointer to the size of the array that holds output of the Multiple Imputation method
<i>simul_missing_vals</i>	float* for vslsSSEditMissingValues double* for vsldsSSEditMissingValues	Pointer to the array of size $k*m$, where k is the total number of missing values, and m is number of copies of missing values. The array holds m sets of simulated missing values for the matrix of observations.
<i>estimates_n</i>	const MKL_INT*	Pointer to the number of estimates to be returned by the routine
<i>estimates</i>	float* for vslsSSEditMissingValues double* for vsldsSSEditMissingValues	Pointer to the array that holds estimates of the mean and a variance-covariance matrix.

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vslsSSEditMissingValues` routine uses values passed as parameters of the routine to replace pointers to the number and the array of the method parameters, pointers to the number and the array of initial mean/variance-covariance estimates, the pointer to the number and the array of prior parameters, pointers to the number and the array of simulated missing values, and pointers to the number and the array of the intermediate mean/covariance estimates. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Before you call the Summary Statistics routines to process missing values, preprocess the dataset and denote missing observations with one of the following predefined constants:

- `VSL_SS_SNAN`, if the dataset is stored in single precision floating-point arithmetic
- `VSL_SS_DNAN`, if the dataset is stored in double precision floating-point arithmetic

Intel® oneAPI Math Kernel Library (oneMKL) provides the `VSL_SS_METHOD_MI` method to support missing values in the dataset based on the Multiple Imputation (MI) approach described in [Schafer97]. The following components support Multiple Imputation:

- Expectation Maximization (EM) algorithm to compute the start point for the Data Augmentation (DA) procedure
- DA function

NOTE

The DA component of the MI procedure is simulation-based and uses the `VSL_BRNG_MCG59` basic random number generator with predefined `seed = 250` and the Gaussian distribution generator (`ICDFmethod`) available in Intel® oneAPI Math Kernel Library (oneMKL) [Gaussian].

Pack the parameters of the MI algorithm into the `params` array. Table "Structure of the Array of MI Parameters" describes the `params` structure.

Structure of the Array of MI Parameters

Array Position	Algorithm Parameter	Description
0	<code>em_iter_num</code>	Maximal number of iterations for the EM algorithm. By default, this value is 50.
1	<code>da_iter_num</code>	Maximal number of iterations for the DA algorithm. By default, this value is 30.
2	<code>ε</code>	Stopping criterion for the EM algorithm. The algorithm terminates if the maximal module of the element-wise difference between the previous and current parameter values is less than <code>ε</code> . By default, this value is 0.001.
3	<code>m</code>	Number of sets to impute
4	<code>missing_vals_num</code>	Total number of missing values in the datasets

You can also pass initial estimates into the EM algorithm by packing both the vector of means and the variance-covariance matrix as a one-dimensional array `init_estimates`. The size of the array should be at least $p + p(p + 1)/2$. For $i=0, \dots, p-1$, the `init_estimates[i]` array contains the initial estimate of means. The remaining positions of the array are occupied by the upper triangular part of the variance-covariance matrix.

If you provide no initial estimates for the EM algorithm, the editor uses the default values, that is, the vector of zero means and the unitary matrix as a variance-covariance matrix. You can also pass `prior` parameters for μ and Σ into the library: μ_0 , τ , m , and Λ^{-1} . Pack these parameters as a one-dimensional array `prior` with a size of at least

$$(p^2 + 3p + 4)/2.$$

The storage format is as follows:

- `prior[0], ..., prior[p-1]` contain the elements of the vector μ_0 .
- `prior[p]` contains the parameter τ .
- `prior[p+1]` contains the parameter m .
- The remaining positions are occupied by the upper-triangular part of the inverted matrix Λ^{-1} .

If you provide no `prior` parameters, the editor uses their default values:

- The array of p zeros is used as μ_0 .
- τ is set to 0.
- m is set to p .
- The zero matrix is used as an initial approximate of Λ^{-1} .

The `EditMissingValues` editor returns m sets of imputed values and/or a sequence of parameter estimates drawn during the DA procedure.

The editor returns the imputed values as the `simul_missing_vals` array. The size of the array should be sufficient to hold m sets each of the `missing_vals_num` size, that is, at least $m \times \text{missing_vals_num}$ in total. The editor packs the imputed values one by one in the order of their appearance in the matrix of observations.

For example, consider a task of dimension 4. The total number of observations n is 10. The second observation vector misses variables 1 and 2, and the seventh observation vector lacks variable 1. The number of sets to impute is $m=2$. Then, `simul_missing_vals[0]` and `simul_missing_vals[1]` contains the first and the second points for the second observation vector, and `simul_missing_vals[2]` holds the first point for the seventh observation. Positions 3, 4, and 5 are formed similarly.

To estimate convergence of the DA algorithm and choose a proper value of the number of DA iterations, request the sequence of parameter estimates that are produced during the DA procedure. The editor returns the sequence of parameters as a single array. The size of the array is

$$m \times \text{da_iter_num} \times (p + (p^2 + p) / 2)$$

where

- m is the number of sets of values to impute.
- `da_iter_num` is the number of DA iterations.
- The value $p + (p^2 + p) / 2$ determines the size of the memory to hold one set of the parameter estimates.

In each set of the parameters, the vector of means occupies the first p positions and the remaining $(p^2 + p) / 2$ positions are intended for the upper triangular part of the variance-covariance matrix.

Upon successful generation of m sets of imputed values, you can place them in cells of the data matrix with missing values and use the Summary Statistics routines to analyze and get estimates for each of the m complete datasets.

NOTE

Intel® oneAPI Math Kernel Library (oneMKL) implementation of the MI algorithm rewrites cells of the dataset that contain the `VSL_SS_SNAN/VSL_SS_DNAN` values. If you want to use the Summary Statistics routines to process the data with missing values again, mask the positions of the empty cells.

See additional details of the algorithm usage model in the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [[SS Notes](#)].

vsISSEditCorParameterization

Modifies pointers to arrays related to the algorithm of correlation matrix parameterization.

Syntax

```
status = vsIsSEditCorParameterization(task, cor, cor_storage, pcor, pcor_storage);
status = vsLdSEditCorParameterization(task, cor, cor_storage, pcor, pcor_storage);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLSTaskPtr	Descriptor of the task
<i>cor</i>	const float* for vslsSSEditCorParameterization const double* for vsldSSEditCorParameterization	Pointer to the correlation matrix
<i>cor_storage</i>	const MKL_INT*	Pointer to the storage format of the correlation matrix
<i>pcor</i>	float* for vslsSSEditCorParameterization double* for vsldSSEditCorParameterization	Pointer to the parameterized correlation matrix
<i>por_storage</i>	const MKL_INT*	Pointer to the storage format of the parameterized correlation matrix

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vslsSSEditCorParameterization` routine uses values passed as parameters of the routine to replace pointers to the correlation matrix, pointers to the correlation matrix storage format, a pointer to the parameterized correlation matrix, and a pointer to the parameterized correlation matrix storage format. See [Table "Storage formats of a variance-covariance/correlation matrix"](#) for possible values of the *cor_storage* and *pcor_storage* parameters. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Summary Statistics Task Computation Routines

Task computation routines calculate statistical estimates on the data provided and parameters held in the task descriptor. After you create the task and initialize its parameters, you can call the computation routines as many times as necessary. [Table "Summary Statistics Estimates Obtained with `vslSSCompute` Routine"](#) lists the respective statistical estimates.

NOTE

The Summary Statistics computation routines do not signal floating-point errors, such as overflow or gradual underflow, or operations with NaNs (except for the missing values in the observations).

Summary Statistics Estimates Obtained with vsISSCompute Routine

Estimate	Support of Observations Available in Blocks	Description
VSL_SS_MEAN	Yes	Computes the array of means.
VSL_SS_SUM	Yes	Computes the array of sums.
VSL_SS_2R_MOM	Yes	Computes the array of the 2 nd order raw moments.
VSL_SS_2R_SUM	Yes	Computes the array of raw sums of the 2 nd order.
VSL_SS_3R_MOM	Yes	Computes the array of the 3 rd order raw moments.
VSL_SS_3R_SUM	Yes	Computes the array of raw sums of the 3 rd order.
VSL_SS_4R_MOM	Yes	Computes the array of the 4 th order raw moments.
VSL_SS_4R_SUM	Yes	Computes the array of raw sums of the 4 th order.
VSL_SS_2C_MOM	Yes	Computes the array of the 2 nd order central moments.
VSL_SS_2C_SUM	Yes	Computes the array of central sums of the 2 nd order.
VSL_SS_3C_MOM	Yes	Computes the array of the 3 rd order central moments.
VSL_SS_3C_SUM	Yes	Computes the array of central sums of the 3 rd order.
VSL_SS_4C_MOM	Yes	Computes the array of the 4 th order central moments.
VSL_SS_4C_SUM	Yes	Computes the array of central sums of the 4 th order.
VSL_SS_KURTOSIS	Yes	Computes the array of kurtosis values.
VSL_SS_SKEWNESS	Yes	Computes the array of skewness values.
VSL_SS_MIN	Yes	Computes the array of minimum values.
VSL_SS_MAX	Yes	Computes the array of maximum values.
VSL_SS_VARIATION	Yes	Computes the array of variation coefficients.
VSL_SS_COV	Yes	Computes a covariance matrix.
VSL_SS_COR	Yes	Computes a correlation matrix. The main diagonal of the correlation matrix holds variances of the random vector components.
VSL_SS_CP	Yes	Computes a cross-product matrix.
VSL_SS_POOLED_COV	No	Computes a pooled covariance matrix.

Estimate	Support of Observations Available in Blocks	Description
VSL_SS_POOLED_MEAN	No	Computes an array of pooled means.
VSL_SS_GROUP_COV	No	Computes group covariance matrices.
VSL_SS_GROUP_MEAN	No	Computes group means.
VSL_SS_QUANTS	No	Computes quantiles.
VSL_SS_ORDER_STATS	No	Computes order statistics.
VSL_SS_ROBUST_COV	No	Computes a robust covariance matrix.
VSL_SS_OUTLIERS	No	Detects outliers in the dataset.
VSL_SS_PARTIAL_COV	No	Computes a partial covariance matrix.
VSL_SS_PARTIAL_COR	No	Computes a partial correlation matrix.
VSL_SS_MISSING_VALS	No	Supports missing values in datasets.
VSL_SS_PARAMTR_COR	No	Computes a parameterized correlation matrix.
VSL_SS_STREAM_QUANTS	Yes	Computes quantiles for streaming data.
VSL_SS_MDAD	No	Computes median absolute deviation.
VSL_SS_MNAD	No	Computes mean absolute deviation.
VSL_SS_SORTED_OBSERV	No	Sorts the dataset by the components of the random vector ξ .

Table "Summary Statistics Computation Method" lists estimate calculation methods supported by Intel® oneAPI Math Kernel Library (oneMKL). See the *Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics Application Notes* document [[SS Notes](#)] for a detailed description of the methods.

Summary Statistics Computation Method

Method	Description
VSL_SS_METHOD_FAST	Fast method for calculation of the estimates: <ul style="list-style-type: none"> raw/central moments/sums, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix min/max/quantile/order statistics partial variance-covariance median/mean absolute deviation
VSL_SS_METHOD_FAST_USER_MEAN	Fast method for calculation of the estimates given user-defined mean: <ul style="list-style-type: none"> central moments/sums of 2-4 order, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix, mean absolute deviation
VSL_SS_METHOD_1PASS	One-pass method for calculation of estimates: <ul style="list-style-type: none"> raw/central moments/sums, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix

Method	Description
	<ul style="list-style-type: none"> pooled/group covariance matrix
VSL_SS_METHOD_TBS	TBS method for robust estimation of covariance and mean
VSL_SS_METHOD_BACON	BACON method for detection of multivariate outliers
VSL_SS_METHOD_MI	Multiple imputation method for support of missing values
VSL_SS_METHOD_SD	Spectral decomposition method for parameterization of a correlation matrix
VSL_SS_METHOD_SQUANTS_ZW	Zhang-Wang (ZW) method for quantile estimation for streaming data
VSL_SS_METHOD_SQUANTS_ZW_FAST	Fast ZW method for quantile estimation for streaming data
VSL_SS_METHOD_RADIX	Radix method for dataset sorting

You can calculate all requested estimates in one call of the routine. For example, to compute a kurtosis and covariance matrix using a fast method, pass a combination of the pre-defined parameters into the `Compute` routine as shown in the example below:

```
...
method = VSL_SS_METHOD_FAST;
task_params = VSL_SS_KURTOSIS|VSL_SS_COV;
...
status = vsldSSCompute( task, task_params, method );
```

To compute statistical estimates for the next block of observations, you can do one of the following:

- copy the observations to memory, starting with the address available to the task
- use one of the appropriate [Editors](#) to modify the pointer to the new dataset in the task.

The library does not detect your changes of the dataset and computed statistical estimates. To obtain statistical estimates for a new matrix, change the observations and initialize relevant arrays. You can follow this procedure to compute statistical estimates for observations that come in portions. See [Table "Summary Statistics Estimates Obtained with `vsldSSCompute` Routine"](#) for information on such observations supported by the Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics estimators.

To modify parameters of the task using the Task Editors, set the address of the targeted matrix of the observations or change the respective vector component indices. After you complete editing the task parameters, you can compute statistical estimates in the modified environment.

If the task completes successfully, the computation routine returns the zero status code. If an error is detected, the computation routine returns an error code. In particular, an error status code is returned in the following cases:

- the task pointer is `NULL`
- memory allocation has failed
- the calculation has failed for some other reason

NOTE

You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

vsldSSCompute

Computes Summary Statistics estimates.

Syntax

```
status = vslsSSCompute(task, estimates, method);
```

```
status = vsldSSCompute(task, estimates, method);
```

Include Files

- `mkh.h`

Input Parameters

Name	Type	Description
<i>task</i>	VSLSSTaskPtr	Descriptor of the task
<i>estimates</i>	const MKL_INT64	List of statistical estimates to compute
<i>method</i>	const MKL_INT	Method to be used in calculations

Output Parameters

Name	Type	Description
<i>status</i>	int	Current status of the task

Description

The `vslSSCompute` routine calculates statistical estimates passed as the *estimates* parameter using the algorithms passed as the *method* parameter of the routine. The computations are done in the context of the task descriptor that contains pointers to all required and optional, if necessary, properly initialized arrays. In one call of the function, you can compute several estimates using proper methods for their calculation. See [Table "Summary Statistics Estimates Obtained with Compute Routine"](#) for the list of the estimates that you can calculate with the `vslSSCompute` routine. See [Table "Summary Statistics Computation Methods"](#) for the list of possible values of the *method* parameter.

To initialize single or double precision version task parameters, use the single (`vslsscompute`) or double (`vsldsscompute`) version of the editor, respectively. To initialize parameters of the integer type, use an integer version of the editor (`vslisscompute`).

NOTE

Requesting a combination of the `VSL_SS_MISSING_VALS` value and any other estimate parameter in the `Compute` function results in processing only the missing values.

Application Notes

Be aware that when computing a correlation matrix, the `vslSSCompute` routine allocates an additional array for each thread which is running the task. If you are running on a large number of threads `vslSSCompute` might consume large amounts of memory.

When calculating covariance, correlation, or cross product, the number of bytes of memory required is at least $(P*P*T + P*T)*b$, where P is the dimension of the task or number of variables, T is the number of threads, and b is the number of bytes required for each unit of data. If observation is weighted and the method is `VSL_SS_METHOD_FAST`, then the memory required is at least $(P*P*T + P*T + N*P)*b$, where N is the number of observations.

Summary Statistics Task Destructor

Task destructor is the `vslSSDeleteTask` routine intended to delete task objects and release memory.

`vslSSDeleteTask`

Destroys the task object and releases the memory.

Syntax

```
status = vslSSDeleteTask(&task);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>VSLSSTaskPtr*</code>	Descriptor of the task to destroy

Output Parameters

Name	Type	Description
<code>status</code>	<code>int</code>	Sets to <code>VSL_STATUS_OK</code> if the task is deleted; otherwise a non-zero code is returned.

Description

The `vslSSDeleteTask` routine deletes the task descriptor object, releases the memory allocated for the structure, and sets the task pointer to `NULL`. If `vslSSDeleteTask` fails to delete the task successfully, it returns an error code.

NOTE

Call of the destructor with the `NULL` pointer as the parameter results in termination of the function with no system crash.

Summary Statistics Usage Examples

The following examples show various standard operations with Summary Statistics routines.

Calculating Fixed Estimates for Fixed Data

The example shows recurrent calculation of the same estimates with a given set of variables for the complete life cycle of the task in the case of a variance-covariance matrix. The set of vector components to process remains unchanged, and the data comes in blocks. Before you call the `vslSSCompute` routine, initialize pointers to arrays for mean and covariance and set buffers.

```
...
double w[2];
double indices[DIM] = {1, 0, 1};

/* calculating mean for 1st and 3d random vector components */
```



```

/* Initialize parameters of the task */
p = DIM;
n = N;

xstorage   = VSL_SS_MATRIX_STORAGE_ROWS;
covstorage = VSL_SS_MATRIX_STORAGE_FULL;

w[0] = 0.0; w[1] = 0.0;

for ( i = 0; i < p; i++ ) mean[i] = 0.0;
for ( i = 0; i < p*p; i++ ) cov[i] = 0.0;

status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, indices );

status = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, w );
status = vsldSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );

```

You can process data arrays that come in blocks as follows:

```

for ( i = 0; i < num_of_blocks; i++ )
{
    status = vsldSSCompute( task, VSL_SS_COV, VSL_SS_METHOD_FAST );
    /* Read new data block into array x */
}
...

```

Calculating Different Estimates for Variable Data

The context of your calculation may change in the process of data analysis. The example below shows the data that comes in two blocks. You need to estimate a covariance matrix for the complete data, and the third central moment for the second block of the data using the weights that were accumulated for the previous datasets. The second block of the data is stored in another array. You can proceed as follows:

```

/* Set parameters for the task */
p = DIM;
n = N;
xstorage   = VSL_SS_MATRIX_STORAGE_ROWS;
covstorage = VSL_SS_MATRIX_STORAGE_FULL;

w[0] = 0.0; w[1] = 0.0;

for ( i = 0; i < p; i++ ) mean[i] = 0.0;
for ( i = 0; i < p*p; i++ ) cov[i] = 0.0;

/* Create task */
status = vsldSSNewTask( &task, &p, &n, &xstorage, x1, 0, indices );

/* Initialize the task parameters */
status = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, w );
status = vsldSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );

/* Calculate covariance for the x1 data */
status = vsldSSCompute( task, VSL_SS_COV, VSL_SS_METHOD_FAST );

/* Initialize array of the 3d central moments and pass the pointer to the task */
for ( i = 0; i < p; i++ ) c3_m[i] = 0.0;

/* Modify task context */

```

```

status = vsldSSeditTask( task, VSL_SS_ED_3C_MOM, c3_m );
status = vsldSSeditTask( task, VSL_SS_ED_OBSERV, x2 );

/* Calculate covariance for the x1 & x2 data block */
/* Calculate the 3d central moment for the 2nd data block using earlier accumulated weight */
status = vsldSScompute(task, VSL_SS_COV|VSL_SS_3C_MOM, VSL_SS_METHOD_FAST );
...
status = vsldSSdeleteTask( &task );

```

Similarly, you can modify indices of the variables to be processed for the next data block.

Summary Statistics Mathematical Notation and Definitions

The following notations are used in the mathematical definitions and the description of the Intel® oneAPI Math Kernel Library (oneMKL) Summary Statistics functions.

Matrix and Weights of Observations

For a random p -dimensional vector $\xi = (\xi_1, \dots, \xi_i, \dots, \xi_p)$, this manual denotes the following:

- $(X)_i = (x_{ij})_{j=1..n}$ is the result of n independent observations for the i -th component ξ_i of the vector ξ .
- The two-dimensional array $X = (x_{ij})_{n \times p}$ is the matrix of observations.
- The column $[X]_j = (x_{ij})_{i=1..p}$ of the matrix X is the j -th observation of the random vector ξ .

Each observation $[X]_j$ is assigned a non-negative weight w_j , where

- The vector $(w_j)_{j=1..n}$ is a vector of weights corresponding to n observations of the random vector ξ .

$$W = \sum_{i=1}^n w_i$$

is the accumulated weight corresponding to observations X .

Vector of sample means

$$M(X) = (M_1(X), \dots, M_p(X)) \text{ with } M_i(X) = \frac{1}{w} \sum_{j=1}^n w_j x_{ij}$$

for all $i = 1, \dots, p$.

Vector of sample partial sums

$$S(X) = (S_1(X), \dots, S_p(X)) \text{ with } S_i(X) = \sum_{j=1}^n w_j x_{ij}$$

for all $i = 1, \dots, p$.

Vector of sample variances

$$V(X) = (V_1(X), \dots, V_p(X)) \text{ with } V_i(X) = \frac{1}{B} \sum_{j=1}^n w_j (x_{ij} - M_i(X))^2, B = W - \sum_{j=1}^n w_j^2 / W$$

for all $i = 1, \dots, p$.

Vector of sample raw/algebraic moments of k -th order, $k \geq 1$

$$R^{(k)}(X) = (R_1^{(k)}(X), \dots, R_p^{(k)}(X)) \text{ with } R_i^{(k)}(X) = \frac{1}{W} \sum_{j=1}^n w_j x_{ij}^k$$

for all $i = 1, \dots, p$.

Vector of sample raw/algebraic partial sums of k -th order, $k=2, 3, 4$ (raw/algebraic partial sums of squares/cubes/fourth powers)

$$S^k(X) = (S_1^k(X), \dots, S_p^k(X)) \text{ with } S_i^k(X) = \sum_{j=1}^n w_j x_{ij}^k$$

for all $i = 1, \dots, p$.

Vector of sample central moments of the third and the fourth order

$$C^{(k)}(X) = (C_1^{(k)}(X), \dots, C_p^{(k)}(X)) \text{ with } C_i^{(k)}(X) = \frac{1}{B} \sum_{j=1}^n w_j (x_{ij} - M_i(X))^k, B = \sum_{j=1}^n w_j$$

for all $i = 1, \dots, p$ and $k = 3, 4$.

Vector of sample central partial sums of k -th order, $k=2, 3, 4$ (central partial sums of squares/cubes/fourth powers)

$$S^k(X) = (S_1^k(X), \dots, S_p^k(X)) \text{ with } S_i^k(X) = \sum_{j=1}^n w_j (x_{ij} - S_i(X))^k$$

for all $i = 1, \dots, p$.

Vector of sample excess kurtosis values

$$B(X) = (B_1(X), \dots, B_p(X)) \text{ with } B_i(X) = \frac{C_i^{(4)}(X)}{V_i^2(X)} - 3$$

for all $i = 1, \dots, p$.

Vector of sample skewness values

$$\Gamma(X) = (\Gamma_1(X), \dots, \Gamma_p(X)) \text{ with } \Gamma_i(X) = \frac{C_i^{(3)}(X)}{V_i^{1.5}(X)}$$

for all $i = 1, \dots, p$.

Vector of sample variation coefficients

$$VC(X) = (VC_1(X), \dots, VC_p(X)) \text{ with } VC_i(X) = \frac{V_i^{0.5}(X)}{M_i(X)}$$

for all $i = 1, \dots, p$.

Matrix of order statistics

Matrix $Y = (Y_{ij})_{p \times n}$, in which the i -th row $(Y)_i = (Y_{ij})_{j=1..n}$ is obtained as a result of sorting in the ascending order of row $(X)_i = (X_{ij})_{j=1..n}$ in the original matrix of observations.

Vector of sample minimum values

$$Min(X) = (Min_1(X), \dots, Min_p(X)), \text{ where } Min_i(X) = y_{i1}$$

for all $i = 1, \dots, p$.

Vector of sample maximum values

$Max(X) = (Max_1(X), \dots, Max_p(X))$, where $Max_i(X) = y_{in}$

for all $i = 1, \dots, p$.

Vector of sample median values

$Med(X) = (Med_1(X), \dots, Med_p(X))$, where $Med_i(X) = \begin{cases} y_{i, (n+1)/2}, & \text{if } n \text{ is odd} \\ (y_{i, n/2} + y_{i, n/2+1})/2, & \text{if } n \text{ is even} \end{cases}$

for all $i = 1, \dots, p$.

Vector of sample median absolute deviations

$MDAD(X) = (MDAD_1(X), \dots, MDAD_p(X))$, where $MDAD_i(X) = Med_i(Z)$ with $Z = (z_{ij})_{i=1\dots p, j=1\dots n'}$
 $z_{ij} = |x_{ij} - Med_i(X)|$

for all $i = 1, \dots, p$.

Vector of sample mean absolute deviations

$MNAD(X) = (MNAD_1(X), \dots, MNAD_p(X))$, where $MNAD_i(X) = M_i(Z)$ with $Z = (z_{ij})_{i=1\dots p, j=1\dots n'}$
 $z_{ij} = |x_{ij} - M_i(X)|$

for all $i = 1, \dots, p$.

Vector of sample quantile values

For a positive integer number q and k belonging to the interval $[0, q-1]$, point z_i is the k -th q quantile of the random variable ξ_i if $P\{\xi_i \leq z_i\} \geq \beta$ and $P\{\xi_i \leq z_i\} < 1 - \beta$, where

- P is the probability measure.
- $\beta = k/q$ is the quantile order.

The calculation of quantiles is as follows:

$j = [(n-1)\beta]$ and $f = \{(n-1)\beta\}$ as integer and fractional parts of the number $(n-1)\beta$, respectively, and the vector of sample quantile values is

$$Q(X, \beta) = (Q_1(X, \beta), \dots, Q_p(X, \beta))$$

where

$$(Q_i(X, \beta) = y_{i, j+1} + f(y_{i, j+2} - y_{i, j+1}))$$

for all $i = 1, \dots, p$.

Variance-covariance matrix

$$C(X) = (c_{ij}(X))_{p \times p}$$

where

$$c_{ij}(X) = \frac{1}{B} \sum_{k=1}^n w_k (x_{ik} - M_i(X))(x_{jk} - M_j(X)), \quad B = W - \sum_{j=1}^n w_j^2 / W$$

Cross-product matrix (matrix of cross-products and sums of squares)

$$CP(X) = (cp_{ij}(X))_{p \times p}$$

where

$$cp_{ij}(X) = \sum_{k=1}^n w_k (x_{ik} - M_i(X)) (x_{jk} - M_j(X))$$

Pooled and group variance-covariance matrices

The set $N = \{1, \dots, n\}$ is partitioned into non-intersecting subsets

$$G_i, i = 1..g, N = \bigcup_{i=1}^g G_i$$

The observation $[X]_j = (x_{ij})_{i=1..p}$ belongs to the group r if $j \in G_r$. One observation belongs to one group only. The group mean and variance-covariance matrices are calculated similarly to the formulas above:

$$M^{(r)}(X) = (M_1^{(r)}(X), \dots, M_p^{(r)}(X)) \text{ with } M_i^{(r)}(X) = \frac{1}{W^{(r)}} \sum_{j \in G_r} w_j x_{ij}, W^{(r)} = \sum_{j \in G_r} w_j$$

for all $i = 1, \dots, p$,

$$C^{(r)}(X) = (c_{ij}^{(r)}(X))_{p \times p}$$

where

$$c_{ij}^{(r)}(X) = \frac{1}{B^{(r)}} \sum_{k \in G_r} w_k (x_{ik} - M_i^{(r)}(X)) (x_{jk} - M_j^{(r)}(X)), B^{(r)} = W^{(r)} - \sum_{j \in G_r} w_j^2 / W^{(r)}$$

for all $i = 1, \dots, p$ and $j = 1, \dots, p$.

A pooled variance-covariance matrix and a pooled mean are computed as weighted mean over group covariance matrices and group means, correspondingly:

$$M^{pooled}(X) = (M_1^{pooled}(X), \dots, M_p^{pooled}(X)) \text{ with } M_i^{pooled}(X) = \frac{1}{W^{(1)} + \dots + W^{(g)}} \sum_{r=1}^g W^{(r)} M_i^{(r)}(X)$$

for all $i = 1, \dots, p$,

$$C^{pooled}(X) = (c_{ij}^{pooled}(X))_{p \times p}, c_{ij}^{pooled}(X) = \frac{1}{B^{(1)} + \dots + B^{(g)}} \sum_{r=1}^g B^{(r)} c_{ij}^{(r)}(X)$$

for all $i = 1, \dots, p$ and $j = 1, \dots, p$.

Correlation matrix

$$R(X) = (r_{ij}(X))_{p \times p}, \text{ where } r_{ij}(X) = \frac{c_{ij}}{\sqrt{c_{ii}c_{jj}}}$$

for all $i = 1, \dots, p$ and $j = 1, \dots, p$.

Partial variance-covariance matrix

For a random vector ξ partitioned into two components Z and Y , a variance-covariance matrix C describes the structure of dependencies in the vector ξ :

$$C(X) = \begin{pmatrix} C_Z(X) & C_{ZY}(X) \\ C_{YZ}(X) & C_Y(X) \end{pmatrix}.$$

The partial covariance matrix $P(X) = (p_{ij}(X))_{k \times k}$ is defined as

$$P(X) = C_Y(X) - C_{YZ}(X) C_Z^{-1}(X) C_{ZY}(X).$$

where k is the dimension of Y .

Partial correlation matrix

The following is a partial correlation matrix for all $i = 1, \dots, k$ and $j = 1, \dots, k$:

$$RP(X) = \left(rp_{ij}(X) \right)_{k \times k}, \text{ where } rp_{ij}(X) = \frac{p_{ij}(X)}{\sqrt{p_{ii}(X)p_{jj}(X)}}$$

where

- k is the dimension of Y .
- $p_{ij}(X)$ are elements of the partial variance-covariance matrix.

Sorted dataset

Matrix $Y = (y_{ij})_{p \times n}$, in which the i -th row $(Y)_i$ is obtained as a result of sorting in ascending order the row $(X)_i = (x_{ij})_{j=1..n}$ in the original matrix of observations.

Fourier Transform Functions

The general form of the discrete Fourier transform is

$$z_{k_1, k_2, \dots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \dots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1, j_2, \dots, j_d} \exp \left(\delta i 2\pi \sum_{l=1}^d j_l k_l / n_l \right)$$

for $k_l = 0, \dots, n_l-1$ ($l = 1, \dots, d$), where σ is a scale factor, $\delta = -1$ for the forward transform, and $\delta = +1$ for the inverse (backward) transform. In the forward transform, the input (periodic) sequence $\{w_{j_1, j_2, \dots, j_d}\}$ belongs to the set of complex-valued sequences and real-valued sequences. Respective domains for the backward transform are represented by complex-valued sequences and complex-valued conjugate-even sequences.

The Intel® oneAPI Math Kernel Library (oneMKL) provides an interface for computing a discrete Fourier transform through the fast Fourier transform algorithm. Prefixes `Dfti` in function names and `DFTI` in the names of configuration parameters stand for Discrete Fourier Transform Interface.

The manual describes the following implementations of the fast Fourier transform functions available in Intel® oneAPI Math Kernel Library (oneMKL):

- Fast Fourier transform (FFT) functions for single-processor or shared-memory systems (see [FFT Functions](#))
- [Cluster FFT functions](#) for distributed-memory architectures (available only for Intel® 64 architectures)

NOTE

Intel® oneAPI Math Kernel Library (oneMKL) also supports the FFTW3* interfaces to the fast Fourier transform functionality for shared memory paradigm (SMP) systems.

Both FFT and Cluster FFT functions compute an FFT in five steps:

1. Allocate a fresh descriptor for the problem with a call to the `DftiCreateDescriptor` or `DftiCreateDescriptorDM` function. The descriptor captures the configuration of the transform, such as the dimensionality (or rank), sizes, number of transforms, memory layout of the input/output data (defined by strides), and scaling factors. Many of the configuration settings are assigned default values in this call which you might need to modify in your application.
2. Optionally adjust the descriptor configuration with a call to the `DftiSetValue` or `DftiSetValueDM` function as needed. Typically, you must carefully define the data storage layout for an FFT or the data distribution among processes for a Cluster FFT. The configuration settings of the descriptor, such as the default values, can be obtained with the `DftiGetValue` or `DftiGetValueDM` function.

3. Commit the descriptor with a call to the [DftiCommitDescriptor](#) or [DftiCommitDescriptorDM](#) function, that is, make the descriptor ready for the transform computation. Once the descriptor is committed, the parameters of the transform, such as the type and number of transforms, strides and distances, the type and storage layout of the data, and so on, are "frozen" in the descriptor.
4. Compute the transform with a call to the [DftiComputeForward](#)/[DftiComputeBackward](#) or [DftiComputeForwardDM](#)/[DftiComputeBackwardDM](#) functions as many times as needed. Because the descriptor is defined and committed separately, all that the compute functions do is take the input and output data and compute the transform as defined. To modify any configuration parameters for another call to a compute function, use [DftiSetValue](#) followed by [DftiCommitDescriptor](#) ([DftiSetValueDM](#) followed by [DftiCommitDescriptorDM](#)) or create and commit another descriptor.
5. Deallocate the descriptor with a call to the [DftiFreeDescriptor](#) or [DftiFreeDescriptorDM](#) function. This returns the memory internally consumed by the descriptor to the operating system.

All the above functions return an integer status value, which is zero upon successful completion of the operation. You can interpret a non-zero status with the help of the [DftiErrorClass](#) or [DftiErrorMessage](#) function.

The FFT functions support lengths with arbitrary factors. You can improve performance of the Intel® oneAPI Math Kernel Library (oneMKL) FFT if the length of your data vector permits factorization into powers of optimized radices. See the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for specific radices supported efficiently.

NOTE

The FFT functions assume the Cartesian representation of complex data (that is, the real and imaginary parts define a complex number). The Intel® oneAPI Math Kernel Library (oneMKL) Vector Mathematical Functions provide efficient tools for conversion to and from polar representation (see [Example "Conversion from Cartesian to polar representation of complex data"](#) and [Example "Conversion from polar to Cartesian representation of complex data"](#)).

Product and Performance Information

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

Notice revision #20201201

FFT Functions

The fast Fourier transform function library of Intel® oneAPI Math Kernel Library (oneMKL) provides one-dimensional, two-dimensional, and multi-dimensional transforms (of up to seven dimensions) and offers both Fortran and C interfaces for all transform functions.

Table "FFT Functions in Intel® oneAPI Math Kernel Library (oneMKL)" lists FFT functions implemented in Intel® oneAPI Math Kernel Library (oneMKL):

FFT Functions in oneMKL

Function Name	Operation
Descriptor Manipulation Functions	
DftiCreateDescriptor	Allocates the descriptor data structure and initializes it with default configuration values.
DftiCommitDescriptor	Performs all initialization for the actual FFT computation.
DftiFreeDescriptor	Frees memory allocated for a descriptor.
DftiCopyDescriptor	Makes a copy of an existing descriptor.

Function Name	Operation
FFT Computation Functions	
DftiComputeForward	Computes the forward FFT.
DftiComputeBackward	Computes the backward FFT.
Descriptor Configuration Functions	
DftiSetValue	Sets one particular configuration parameter with the specified configuration value.
DftiGetValue	Gets the value of one particular configuration parameter.
Status Checking Functions	
DftiErrorClass	Checks if the status reflects an error of a predefined class.
DftiErrorMessage	Translates the numeric value of an error status into a message.

FFT Interface

The Intel® oneAPI Math Kernel Library (oneMKL) FFT functions are provided with the Fortran and C interfaces. The materials presented in this section assume the availability of native complex types in C as they are specified in C9X.

To use the FFT functions, you need to include `mkl_dfti.h` in your C code.

The C interface provides the `DFTI_DESCRIPTOR_HANDLE` type, named constants of two enumeration types `DFTI_CONFIG_PARAM` and `DFTI_CONFIG_VALUE`, and functions, some of which accept different numbers of input arguments.

NOTE

The current version of the library may not support some of the FFT functions or functionality. You can find the complete list of the implementation-specific exceptions in the Intel® oneAPI Math Kernel Library (oneMKL) Release Notes.

For the main categories of Intel® oneAPI Math Kernel Library (oneMKL) FFT functions, see [FFT Functions](#).

Computing an FFT

You can find code examples that compute transforms in the [Fourier Transform Functions Code Examples](#).

Usually you can compute an FFT by five function calls (refer to the [usage model](#) for details). A single data structure, the descriptor, stores configuration parameters that can be changed independently.

The descriptor data structure, when created, contains information about the length and domain of the FFT to be computed, as well as the setting of several configuration parameters. Default settings for some of these parameters are as follows:

- Scale factor: none (that is, $\sigma = 1$)
- Number of data sets: one
- Data storage: contiguous
- Placement of results: in-place (the computed result overwrites the input data)

The default settings can be changed one at a time through the function [DftiSetValue](#) as illustrated in [Example "Changing Default Settings \(C\)"](#).

Configuration Settings

Each of the configuration parameters is identified by a named constant in the `MKL_DFTI` module. These named constants have the enumeration type `DFTI_CONFIG_PARAM` and are declared in the `mkl_dfti.h` header file.

Though exposed in that header file, the configuration parameters `DFTI_FWD_DISTANCE` and `DFTI_BWD_DISTANCE` are specific to the DPC++ DFT routines (see the [Data Parallel C++ Developer Reference](#)): their use in another context is neither supported nor currently enabled. All other Intel® oneAPI Math Kernel Library (oneMKL) FFT configuration parameters are readable. Some of them are read-only, while others can be set using the [DftiCreateDescriptor](#) or [DftiSetValue](#) function.

Values of the configuration parameters fall into the following groups:

- Values that have native data types. For example, the number of simultaneous transforms requested has an integer value, while the scale factor for a forward transform is a floating-point number.
- Values that are discrete in nature and are provided in the `MKL_DFTI` module as named constants. For example, the domain of the forward transform requires values to be named constants. The named constants for configuration values have the enumeration type `DFTI_CONFIG_VALUE`.

The [Table "Configuration Parameters"](#) summarizes the information on configuration parameters, along with their types and values. For more details of each configuration parameter, see the subsection describing this parameter.

Configuration Parameters

Configuration Parameter	Type/Value	Comments
<i>Most common configuration parameters, no default, must be set explicitly by <code>DftiCreateDescriptor</code></i>		
<code>DFTI_PRECISION</code>	Named constant <code>DFTI_SINGLE</code> or <code>DFTI_DOUBLE</code>	Precision of the computation.
<code>DFTI_FORWARD_DOMAIN</code>	Named constant <code>DFTI_COMPLEX</code> or <code>DFTI_REAL</code>	Type of the transform.
<code>DFTI_DIMENSION</code>	Integer scalar	Dimension of the transform.
<code>DFTI_LENGTHS</code>	Integer scalar/array	Lengths of each dimension.
<i>Common configuration parameters, settable by <code>DftiSetValue</code></i>		
<code>DFTI_PLACEMENT</code>	Named constant <code>DFTI_INPLACE</code> or <code>DFTI_NOT_INPLACE</code>	Defines whether the result overwrites the input data. Default value: <code>DFTI_INPLACE</code> .
<code>DFTI_FORWARD_SCALE</code>	Floating-point scalar	Scale factor for the forward transform. Default value: 1.0. Precision of the value should be the same as defined by <code>DFTI_PRECISION</code> .
<code>DFTI_BACKWARD_SCALE</code>	Floating-point scalar	Scale factor for the backward transform. Default value: 1.0. Precision of the value should be the same as defined by <code>DFTI_PRECISION</code> .
<code>DFTI_NUMBER_OF_USER_THREADS</code>	Integer scalar	This configuration parameter is no longer used and kept for compatibility with previous versions of Intel® oneAPI Math Kernel Library (oneMKL).

Configuration Parameter	Type/Value	Comments
DFTI_THREAD_LIMIT	Integer scalar	Limits the number of threads for the DftiComputeForward and DftiComputeBackward . Default value: 0.
DFTI_DESCRIPTOR_NAME	Character string	Assigns a name to a descriptor. Assumed length of the string is DFTI_MAX_NAME_LENGTH . Default value: empty string.
<i>Data layout configuration parameters for single and multiple transforms. Settable by DftiSetValue</i>		
DFTI_INPUT_STRIDES	Integer array	Defines the input data layout. NOTE The default strides are set during creation of the descriptor based on the desired dimension and lengths. For more details, see DFTI_INPUT_STRIDES , DFTI_OUTPUT_STRIDES .
DFTI_OUTPUT_STRIDES	Integer array	Defines the output data layout. NOTE The default strides are set during creation of the descriptor based on the desired dimension and lengths. For more details, see DFTI_INPUT_STRIDES , DFTI_OUTPUT_STRIDES .
DFTI_NUMBER_OF_TRANSFORMS	Integer scalar	Number of transforms. Default value: 1.
DFTI_INPUT_DISTANCE	Integer scalar	Defines the distance between input data sets for multiple transforms. Default value: 0.
DFTI_OUTPUT_DISTANCE	Integer scalar	Defines the distance between output data sets for multiple transforms. Default value: 0.
DFTI_COMPLEX_STORAGE	Named constant DFTI_COMPLEX_COMPLEX X or DFTI_REAL_REAL	Defines whether the real and imaginary parts of data for a complex transform are interleaved in one array or split in two arrays. Default value: DFTI_COMPLEX_COMPLEX .
DFTI_REAL_STORAGE	Named constant DFTI_REAL_REAL	Defines how real data for a real transform is stored. Only the DFTI_REAL_REAL value is supported.
DFTI_CONJUGATE_EVEN_STORAGE	Named constant DFTI_COMPLEX_COMPLEX X or DFTI_COMPLEX_REAL	Defines whether the complex data in the backward domain of a real transform is stored as complex elements or as real elements.

Configuration Parameter	Type/Value	Comments
		DFTI_COMPLEX_REAL is supported only for 1D transforms. The default value is DFTI_COMPLEX_COMPLEX.
DFTI_PACKED_FORMAT	Named constant DFTI_CCE_FORMAT, DFTI_CCS_FORMAT, DFTI_PACK_FORMAT, or DFTI_PERM_FORMAT	Defines the layout for the elements of the conjugate-even sequence in the backward domain of the real transform (in association with the configuration parameter DFTI_CONJUGATE_EVEN_STORAGE). The default value is DFTI_CCE_FORMAT.
NOTE Transforms greater than 1D support only DFTI_CCE_FORMAT.		
<i>Advanced configuration parameters, settable by DftiSetValue</i>		
DFTI_WORKSPACE	Named constant DFTI_ALLOW or DFTI_AVOID	Defines whether the library should prefer algorithms using additional memory. Default value: DFTI_ALLOW.
DFTI_ORDERING	Named constant DFTI_ORDERED or DFTI_BACKWARD_SCRAMBLED	Defines whether the result of a complex transform is ordered or permuted. Default value: DFTI_ORDERED.
DFTI_DESTROY_INPUT	Named constant DFTI_ALLOW or DFTI_AVOID	Defines whether the input data may be overwritten for out-of-place transforms. Default value: DFTI_AVOID.
<i>Read-Only configuration parameters</i>		
DFTI_COMMIT_STATUS	Named constant DFTI_UNCOMMITTED or DFTI_COMMITTED	Readiness of the descriptor for computation.
DFTI_VERSION	String	Version of Intel® oneAPI Math Kernel Library (oneMKL). Assumed length of the string is DFTI_VERSION_LENGTH.

See Also

Configuring and Computing an FFT in C/C++

DFTI_PRECISION

The configuration parameter DFTI_PRECISION denotes the floating-point precision in which the transform is to be carried out. A setting of DFTI_SINGLE stands for single precision, and a setting of DFTI_DOUBLE stands for double precision. The data must be presented in this precision, the computation is carried out in this precision, and the result is delivered in this precision.

DFTI_PRECISION does not have a default value. Set it explicitly by calling the DftiCreateDescriptor function.

To better understand configuration of the precision of transforms, refer to these examples in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

```
./examples/dftc/source/basic_sp_complex_dft_1d.c
```

`./examples/dftc/source/basic_dp_complex_dft_1d.c`

See Also

`DFTI_FORWARD_DOMAIN`

`DFTI_DIMENSION`, `DFTI_LENGTHS`

`DftiCreateDescriptor`

DFTI_FORWARD_DOMAIN

The general form of a discrete Fourier transform is

$$z_{k_1, k_2, \dots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \dots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1, j_2, \dots, j_d} \exp\left(\delta i 2\pi \sum_{l=1}^d j_l k_l / n_l\right)$$

for $k_l = 0, \dots, n_l-1$ ($l = 1, \dots, d$), where σ is a scale factor, $\delta = -1$ for the forward transform, and $\delta = +1$ for the backward transform.

The Intel® oneAPI Math Kernel Library (oneMKL) implementation of the FFT algorithm, used for fast computation of discrete Fourier transforms, supports forward transforms on input sequences of two domains, as specified by the `DFTI_FORWARD_DOMAIN` configuration parameter: general complex-valued sequences (`DFTI_COMPLEX` domain) and general real-valued sequences (`DFTI_REAL` domain). The forward transform maps the forward domain to the corresponding backward domain, as shown in [Table "Correspondence of Forward and Backward Domain"](#).

The conjugate-even domain covers complex-valued sequences with the symmetry property:

$$x(k_1, k_2, \dots, k_d) = \text{conjugate}(x(n_1 - k_1, n_2 - k_2, \dots, n_d - k_d))$$

where the index arithmetic is performed modulo respective size, that is,

$$x(\dots, \text{expr}_s, \dots) \equiv x(\dots, \text{mod}(\text{expr}_s, n_s), \dots),$$

and therefore

$$x(\dots, n_s, \dots) \equiv x(\dots, 0, \dots).$$

Due to this property of conjugate-even sequences, only a part of such sequence is stored in the computer memory, as described in `DFTI_CONJUGATE_EVEN_STORAGE`.

Correspondence of Forward and Backward Domain

Forward Domain	Implied Backward Domain
Complex (<code>DFTI_COMPLEX</code>)	Complex (<code>DFTI_COMPLEX</code>)
Real (<code>DFTI_REAL</code>)	Conjugate-even

`DFTI_FORWARD_DOMAIN` does not have a default value. Set it explicitly by calling the `DftiCreateDescriptor` function.

To better understand usage of the `DFTI_FORWARD_DOMAIN` configuration parameter, you can refer to these examples in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

`./examples/dftc/source/basic_sp_complex_dft_1d.c`

`./examples/dftc/source/basic_sp_real_dft_1d.c`

See Also

`DFTI_PRECISION`

`DFTI_DIMENSION`, `DFTI_LENGTHS`

`DftiCreateDescriptor`

DFTI_DIMENSION, DFTI_LENGTHS

The dimension of the transform is a positive integer value represented in an integer scalar of `MKL_LONG` data type. For a one-dimensional transform, the transform length is specified by a positive integer value represented in an integer scalar of `MKL_LONG` data type. For multi-dimensional (≥ 2) transform, the lengths of each of the dimensions are supplied in an integer array (of `MKL_LONG` data type).

`DFTI_DIMENSION` and `DFTI_LENGTHS` do not have a default value. To set them, use the `DftiCreateDescriptor` function and not the `DftiSetValue` function.

To better understand usage of the `DFTI_DIMENSION` and `DFTI_LENGTHS` configuration parameters, you can refer to basic examples of one-, two-, and three-dimensional transforms in your Intel® oneAPI Math Kernel Library (oneMKL) directory. Naming conventions for the examples are self-explanatory. For example, refer to these examples of single-precision two-dimensional transforms:

```
./examples/dftc/source/basic_sp_real_dft_2d.c
./examples/dftc/source/basic_sp_complex_dft_2d.c
```

See Also

[DFTI_FORWARD_DOMAIN](#)

[DFTI_PRECISION](#)

[DftiCreateDescriptor](#)

[DftiSetValue](#)

DFTI_PLACEMENT

By default, the computational functions overwrite the input data with the output result. That is, the default setting of the configuration parameter `DFTI_PLACEMENT` is `DFTI_INPLACE`. You can change that by setting it to `DFTI_NOT_INPLACE`.

NOTE

When the configuration parameter is set to `DFTI_NOT_INPLACE`, the input and output data sets must have no common elements.

To better understand usage of the `DFTI_PLACEMENT` configuration parameter, see this example in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

```
./examples/dftc/source/config_placement.c
```

See Also

[DftiSetValue](#)

DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE

The forward transform and backward transform are each associated with a scale factor σ of its own having the default value of 1. You can specify the scale factors using one or both of the configuration parameters `DFTI_FORWARD_SCALE` and `DFTI_BACKWARD_SCALE`. For example, for a one-dimensional transform of length n , you can use the default scale of 1 for the forward transform and set the scale factor for the backward transform to be $1/n$, thus making the backward transform the inverse of the forward transform.

Set the scale factor configuration parameter using a real floating-point data type of the same precision as the value for `DFTI_PRECISION`.

NOTE

For inquiry of the scale factor with the `DftiGetValue` function, the `config_val` parameter must have the same floating-point precision as the descriptor.

See Also

[DftiSetValue](#)

[DFTI_PRECISION](#)

[DftiGetValue](#)

DFTI_NUMBER_OF_USER_THREADS

The `DFTI_NUMBER_OF_USER_THREADS` configuration parameter is no longer used and kept for compatibility with previous versions of Intel® oneAPI Math Kernel Library (oneMKL).

See Also

[DftiSetValue](#)

DFTI_THREAD_LIMIT

In some situations you may need to limit the number of threads that the `DftiComputeForward` and `DftiComputeBackward` functions use. For example, if more than one thread calls Intel® oneAPI Math Kernel Library (oneMKL), it might be important that the thread calling these functions does not oversubscribe computing resources (CPU cores). Similarly, a known limit of the maximum number of threads to be used in computations might help the `DftiCommitDescriptor` function to select a more optimal computation method.

Set the parameter `DFTI_THREAD_LIMIT` as follows:

- To a positive number, to specify the maximum number of threads to be used by the compute functions.
- To zero (the default value), to use the maximum number of threads permitted in Intel® oneAPI Math Kernel Library (oneMKL) FFT functions. See "Techniques to Set the Number of Threads" in the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for more information.

On an attempt to set a negative value, the `DftiSetValue` function returns an error and does not update the descriptor.

The value of the `DFTI_THREAD_LIMIT` configuration parameter returned by the `DftiGetValue` function is defined as follows:

- 1 if Intel® oneAPI Math Kernel Library (oneMKL) runs in the sequential mode
- Depends of the commit status of the descriptor if Intel® oneAPI Math Kernel Library (oneMKL) runs in a threaded mode:

Commit Status	Value
Not committed	The value of <code>DFTI_THREAD_LIMIT</code> set in a previous call to the <code>DftiSetValue</code> function or the default value
Committed	The upper limit on the number of threads used by the <code>DftiComputeForward</code> and <code>DftiComputeBackward</code> functions

To better understand usage of the `DFTI_THREAD_LIMIT` configuration parameter, see this example in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

```
./examples/dftc/source/config_thread_limit.c
```

See Also

[DftiGetValue](#)

[DftiSetValue](#)

[DftiCommitDescriptor](#)

[DftiComputeForward](#)

[DftiComputeBackward](#)

Threading Control Functions

[DFTI_COMMIT_STATUS](#)

DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES

The FFT interface provides configuration parameters that define the layout of multidimensional data in the computer memory. For d -dimensional data set X defined by dimensions $N_1 \times N_2 \times \dots \times N_d$, the layout describes where a particular element $X(k_1, k_2, \dots, k_d)$ of the data set is located. The memory address of the element $X(k_1, k_2, \dots, k_d)$ is expressed by the formula,

address of $X(k_1, k_2, \dots, k_d)$ = the address stored in the pointer supplied to the compute function + $(s_0 + k_1*s_1 + k_2*s_2 + \dots + k_d*s_d) * u$,

Where u is the number of bytes per element of the desired precision for the assumed data type in the corresponding domain (see Table "Assumed Element Types of the Input/Output Data" below), where s_0 is the displacement, and s_1, \dots, s_d are generalized strides. The configuration parameters `DFTI_INPUT_STRIDES` and `DFTI_OUTPUT_STRIDES` enable you to get and set these values. The configuration value is an array of values (s_0, s_1, \dots, s_d) of `MKL_LONG` data type.

The `DFTI_FORWARD_DOMAIN`, `DFTI_COMPLEX_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the type of the elements as shown in Table "Assumed Element Types of the Input/Output Data":

Assumed Element Types of the Input/Output Data

Descriptor Configuration	Element Type in the Forward Domain	Element Type in the Backward Domain
<code>DFTI_FORWARD_DOMAIN=DFTI_COMPLEX</code> <code>DFTI_COMPLEX_STORAGE=DFTI_COMPLEX_COMPLEX</code>	Complex	Complex
<code>DFTI_FORWARD_DOMAIN=DFTI_COMPLEX</code> <code>DFTI_COMPLEX_STORAGE=DFTI_REAL_REAL</code>	Real	Real
<code>DFTI_FORWARD_DOMAIN=DFTI_REAL</code> <code>DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL</code>	Real	Real
<code>DFTI_FORWARD_DOMAIN=DFTI_REAL</code> <code>DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX</code>	Real	Complex

The `DFTI_INPUT_STRIDES` configuration parameter defines the layout of the input data, while the element type is defined by the forward domain for the `DftiComputeForward` function and by the backward domain for the `DftiComputeBackward` function. The `DFTI_OUTPUT_STRIDES` configuration parameter defines the layout of the output data, while the element type is defined by the backward domain for the `DftiComputeForward` function and by the forward domain for `DftiComputeBackward` function.

NOTE

The `DFTI_INPUT_STRIDES` and `DFTI_OUTPUT_STRIDES` configuration parameters define the layout of input and output data, and not the forward-domain and backward-domain data. If the data layouts in forward domain and backward domain differ, set `DFTI_INPUT_STRIDES` and `DFTI_OUTPUT_STRIDES` explicitly and then commit the descriptor before calling computation functions.

For in-place transforms (`DFTI_PLACEMENT=DFTI_INPLACE`), the configuration set by `DFTI_OUTPUT_STRIDES` is ignored when the element types in the forward and backward domains are the same. If they are different, set `DFTI_OUTPUT_STRIDES` explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the first elements on input and output must coincide in each dimension.

The FFT interface supports both positive and negative stride values. If you use negative strides, set the displacement of the data as follows:

$$s_0 = \sum_{i=1}^d (N_i - 1) \cdot \max(-s_i, 0).$$

The default setting of strides in a general multi-dimensional case assumes that the array that contains the data has no padding. The order of the strides depends on the programming language. For example:

```
MKL_LONG dims[] = { n_d, ..., n_2, n_1 };
DftiCreateDescriptor( &hand, precision, domain, d, dims );
// The above call assumes data declaration:  type X[n_d]...[n_2][n_1]
// Default strides are { 0, n_d-1*...*n_2*n_1, ..., n_2*n_1*1, n_1*1, 1 }
```

Note that in case of a real FFT (`DFTI_FORWARD_DOMAIN=DFTI_REAL`), where different data layouts in the backward domain are available (see [DFTI_PACKED_FORMAT](#)), the default value of the strides is not intuitive for the recommended CCE format (configuration setting `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX`). In case of an *in-place* real transform with the CCE format, set the strides explicitly, as follows:

```
MKL_LONG dims[] = { n_d, ..., n_2, n_1 };
MKL_LONG rstrides[] = { 0, 2*n_d-1*...*n_2*(n_1/2+1), ..., 2*n_2*(n_1/2+1), 2*(n_1/2+1), 1 };
MKL_LONG cstrides[] = { 0, n_d-1*...*n_2*(n_1/2+1), ..., n_2*(n_1/2+1), (n_1/2+1), 1 };
DftiCreateDescriptor( &hand, precision, DFTI_REAL, d, dims );
DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX);
// Set the strides appropriately for forward/backward transform
```

Limitation Note Transforms with the number of points N of a non-unit stride dimension exceeding $2^{(27-p)} - 1$ for N a power-of-two, or $2^{(23-p)} - 1$ for N not a power-of-two, are currently not supported, where $p=0$ for single precision and $p=1$ for double precision. If a descriptor is created (for example, using `DftiCreateDescriptor`) and set (for example, using `DftiSetValue`) to do such a transform, a `DFTI_1D_MEMORY_EXCEEDS_INT32` error is returned at commit time (for example, by `DftiCommitDescriptor`).

To better understand configuration of strides, you can also refer to these examples in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

```
./examples/dftc/source/basic_sp_complex_dft_2d.c
./examples/dftc/source/basic_sp_complex_dft_3d.c
./examples/dftc/source/basic_dp_complex_dft_2d.c
./examples/dftc/source/basic_dp_complex_dft_3d.c
./examples/dftc/source/basic_sp_real_dft_2d.c
./examples/dftc/source/basic_sp_real_dft_3d.c
./examples/dftc/source/basic_dp_real_dft_2d.c
./examples/dftc/source/basic_dp_real_dft_3d.c
```

See Also

[DFTI_FORWARD_DOMAIN](#)

[DFTI_PLACEMENT](#)

FFT Code Examples

[DftiSetValue](#)

[DftiCommitDescriptor](#)

[DftiComputeForward](#)

[DftiComputeBackward](#)

DFTI_NUMBER_OF_TRANSFORMS

If you need to perform a large number of identical FFTs, you can do this in a single call to a `DftiCompute*` function with the value of the `DFTI_NUMBER_OF_TRANSFORMS` configuration parameter equal to the actual number of the transforms. The default value of this parameter is 1. You can set this parameter to a positive integer value of the `MKL_LONG` data type. When setting the number of transforms to a value greater than one, you also need to specify the distance between the input data sets and the distance between the output data sets using one of the `DFTI_INPUT_DISTANCE` and `DFTI_OUTPUT_DISTANCE` configuration parameters or both.

Important

- The data sets to be transformed must not have common elements.
 - All the sets of data must be located within the same memory block.
-

To better understand usage of the `DFTI_NUMBER_OF_TRANSFORMS` configuration parameter, see this example in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

```
./examples/dftc/source/config_number_of_transforms.c
```

See Also

FFT Computation Functions

`DFTI_INPUT_DISTANCE`, `DFTI_OUTPUT_DISTANCE`

`DftiSetValue`

DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE

The FFT interface in Intel® oneAPI Math Kernel Library (oneMKL) enables computation of multiple transforms. To compute multiple transforms, you need to specify the data distribution of the multiple sets of data. The distance between the first data elements of consecutive data sets, `DFTI_INPUT_DISTANCE` for input data or `DFTI_OUTPUT_DISTANCE` for output data, specifies the distribution. The configuration setting is a value of `MKL_LONG` data type.

The default value for both configuration settings is one. You must set this parameter explicitly if the number of transforms is greater than one (see [DFTI_NUMBER_OF_TRANSFORMS](#)).

The distance is counted in elements of the data type defined by the descriptor configuration (rather than by the type of the variable passed to the computation functions). Specifically, the `DFTI_FORWARD_DOMAIN`, `DFTI_COMPLEX_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the type of the elements as shown in [Table "Assumed Element Types of the Input/Output Data"](#).

NOTE

The configuration parameters `DFTI_INPUT_DISTANCE` and `DFTI_OUTPUT_DISTANCE` define the distance within input and output data, and not within the forward-domain and backward-domain data. If the distances in the forward and backward domains differ, set `DFTI_INPUT_DISTANCE` and `DFTI_OUTPUT_DISTANCE` explicitly and then commit the descriptor before calling computation functions.

For in-place transforms (`DFTI_PLACEMENT=DFTI_INPLACE`), the configuration set by `DFTI_OUTPUT_DISTANCE` is ignored when the element types in the forward and backward domains are the same. If they are different, set `DFTI_OUTPUT_DISTANCE` explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the data sets on input and output must coincide.

This example illustrates setting of the `DFTI_INPUT_DISTANCE` configuration parameter:

```
MKL_LONG dims[] = { nd, ..., n2, n1 };
MKL_LONG distance = nd*...*n2*n1;
DftiCreateDescriptor( &hand, precision, DFTI_COMPLEX, d, dims );
DftiSetValue( hand, DFTI_NUMBER_OF_TRANSFORMS, (MKL_LONG)howmany );
DftiSetValue( hand, DFTI_INPUT_DISTANCE, distance );
```

To better understand configuration of the distances, see these code examples in your Intel® oneAPI Math Kernel Library (oneMKL) directory:

`./examples/dftc/source/config_number_of_transforms.c`

See Also

[DFTI_PLACEMENT](#)

[DftiSetValue](#)

[DftiCommitDescriptor](#)

[DftiComputeForward](#)

[DftiComputeBackward](#)

DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE

Depending on the value of the `DFTI_FORWARD_DOMAIN` configuration parameter, the implementation of FFT supports several storage schemes for input and output data (see document [3] for the rationale behind the definition of the storage schemes). The data elements are placed within contiguous memory blocks, defined with generalized strides (see [DFTI_INPUT_STRIDES](#), [DFTI_OUTPUT_STRIDES](#)). For multiple transforms, all sets of data should be located within the same memory block, and the data sets should be placed at the same distance from each other (see [DFTI_NUMBER_OF_TRANSFORMS](#) and [DFTI_INPUT_DISTANCE](#), [DFTI_OUTPUT_DISTANCE](#)).

Tip

Avoid setting up multidimensional arrays with lists of pointers to one-dimensional arrays. Instead use a one-dimensional array with the explicit indexing to access the data elements.

[FFT Examples](#) demonstrate the usage of storage formats.

DFTI_COMPLEX_STORAGE: storage schemes for a complex domain

For the `DFTI_COMPLEX` forward domain, both input and output sequences belong to a complex domain. In this case, the configuration parameter `DFTI_COMPLEX_STORAGE` can have one of the two values:

`DFTI_COMPLEX_COMPLEX` (default) or `DFTI_REAL_REAL`.

NOTE

In the Intel® oneAPI Math Kernel Library (oneMKL)FFT interface, storage schemes for a forward complex domain and the respective backward complex domain are the same.

With `DFTI_COMPLEX_COMPLEX` storage, complex-valued data sequences are referenced by a single complex parameter (array) `AZ` so that a complex-valued element z_{k_1, k_2, \dots, k_d} of the m -th d -dimensional sequence is located at `AZ[m*distance + stride0 + k1*stride1 + k2*stride2+ ... kd*strided]` as a structure consisting of the real and imaginary parts.

This code illustrates the use of the `DFTI_COMPLEX_COMPLEX` storage:

```
complex *AZ = malloc( N1*N2*N3*M * sizeof(AZ[0]) );
MKL_LONG ios[4], iodist; // input/output strides and distance
...
```

```
// on input:  Z{k1,k2,k3,m}
// = AZ[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
status = DftiComputeForward( desc, AZ );
// on output: Z{k1,k2,k3,m}
// = AZ[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
```

With the `DFTI_REAL_REAL` storage, complex-valued data sequences are referenced by two real parameters `AR` and `AI` so that a complex-valued element z_{k_1, k_2, \dots, k_d} of the m -th sequence is computed as $AR[m*distance + stride_0 + k_1*stride_1 + k_2*stride_2 + \dots k_d*stride_d] + \sqrt{-1} * AI[m*distance + stride_0 + k_1*stride_1 + k_2*stride_2 + \dots k_d*stride_d]$.

This code illustrates the use of the `DFTI_REAL_REAL` storage:

```
float *AR = malloc( N1*N2*N3*M * sizeof(AR[0]) );
float *AI = malloc( N1*N2*N3*M * sizeof(AI[0]) );
MKL_LONG ios[4], iodist; // input/output strides and distance
...
// on input:  Z{k1,...,kd,m}
// =  AR[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
// + I*AI[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
status = DftiComputeForward( desc, AR, AI );
// on output: Z{k1,...,kd,m}
// =  AR[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
// + I*AI[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
```

DFTI_REAL_STORAGE: storage schemes for a real domain

The Intel® oneAPI Math Kernel Library (oneMKL) FFT interface supports only one configuration value for this storage scheme: `DFTI_REAL_REAL`. With the `DFTI_REAL_REAL` storage, real-valued data sequences in a real domain are referenced by one real parameter `AR` so that real-valued element of the m -th sequence is located as $AR[m*distance + stride_0 + k_1*stride_1 + k_2*stride_2 + \dots k_d*stride_d]$.

DFTI_CONJUGATE_EVEN_STORAGE: storage scheme for a conjugate-even domain

The Intel® oneAPI Math Kernel Library (oneMKL) FFT interface supports two configuration values for this parameter: `DFTI_COMPLEX_COMPLEX` (default) and `DFTI_COMPLEX_REAL` (for 1D problems only). The conjugate-even symmetry of the data enables storing only about a half of the whole mathematical result, so that one part of it can be directly referenced in the memory while the other part can be reconstructed depending on the selected storage configuration.

With the `DFTI_COMPLEX_COMPLEX` storage, the complex-valued data sequences in the conjugate-even domain are referenced by one complex parameter `AZ` so that a complex-valued element z_{k_1, k_2, \dots, k_d} of the m -th sequence can be referenced or reconstructed as described below.

Consider a d -dimensional real-to-complex transform:

$$Z_{k_1, k_2, \dots, k_d} \equiv \sum_{n_1=0}^{N_1-1} \dots \sum_{n_d=0}^{N_d-1} R_{n_1, n_2, \dots, n_d} e^{\frac{-2\pi i}{N_1} k_1 \cdot n_1} \dots e^{\frac{-2\pi i}{N_d} k_d \cdot n_d}$$

Because the input sequence R is real-valued, the mathematical result Z has conjugate-even symmetry:

$$z_{k_1, k_2, \dots, k_d} = \text{conjugate}(z_{N_1-k_1, N_2-k_2, \dots, N_d-k_d}),$$

where index arithmetic is performed modulo the length of the respective dimension. Obviously, the first element of the result is real-valued:

$$z_{0, 0, \dots, 0} = \text{conjugate}(z_{0, 0, \dots, 0}).$$

For dimensions with even lengths, some of the other elements are real-valued too. For example, if N_s is even, $z_0, 0, \dots, N_s/2, 0, \dots, 0 = \text{conjugate}(z_0, 0, \dots, N_s/2, 0, \dots, 0)$.

With the conjugate-even symmetry, approximately a half of the result suffices to fully reconstruct it. For an arbitrary dimension h , it suffices to store elements $z_{k_1, \dots, k_h, \dots, k_d}$ for the following indices:

- $k_h = 0, \dots, \lfloor N_h/2 \rfloor$
- $k_i = 0, \dots, N_i - 1$, where $i = 1, \dots, d$ and $i \neq h$

The symmetry property enables reconstructing the remaining elements: for $k_h = \lfloor N_h/2 \rfloor + 1, \dots, N_h - 1$. In the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface, the halved dimension is the last dimension.

The following code illustrates usage of the `DFTI_COMPLEX_COMPLEX` storage for a conjugate-even domain:

```
float *AR = malloc( N1*N2*M * sizeof(AR[0]) );
complex *AZ = malloc( N1*(N2/2+1)*M * sizeof(AZ[0]) );
MKL_LONG is[3], os[3], idist, odist; // input and output strides and distance
...
// on input: R{k1,k2,m}
// = AR[is[0] + k1*is[1] + k2*is[2] + m*idist]
status = DftiComputeForward( desc, R, C );
// on output:
// for k2=0...N2/2: Z{k1,k2,m} = AZ[os[0]+k1*os[1]+k2*os[2]+m*odist]
// for k2=N2/2+1...N2-1: Z{k1,k2,m} = conj( AZ[os[0]+(N1-k1)%N1*os[1]
//                                     + (N2-k2)%N2*os[2]+m*odist] )
```

For the backward transform, the input and output parameters and layouts exchange roles: set the strides describing the layout in the backward/forward domain as input/output strides, respectively. For example:

```
...
status = DftiSetValue( desc, DFTI_INPUT_STRIDES, fwd_domain_strides );
status = DftiSetValue( desc, DFTI_OUTPUT_STRIDES, bwd_domain_strides );
status = DftiCommitDescriptor( desc );
status = DftiComputeForward( desc, ... );
...
status = DftiSetValue( desc, DFTI_INPUT_STRIDES, bwd_domain_strides );
status = DftiSetValue( desc, DFTI_OUTPUT_STRIDES, fwd_domain_strides );
status = DftiCommitDescriptor( desc );
status = DftiComputeBackward( desc, ... );
```

Important

For in-place transforms, ensure the first element of the input data has the same location as the first element of the output data for each dimension.

See Also

[DftiSetValue](#)

DFTI_PACKED_FORMAT

The result of the forward transform of real data is a conjugate-even sequence. Due to the symmetry property, only a part of the complex-valued sequence is stored in memory. The combination of the `DFTI_PACKED_FORMAT` and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters defines how the conjugate-even sequence data is packed. If `DFTI_CONJUGATE_EVEN_STORAGE` is set to `DFTI_COMPLEX_COMPLEX` (default), the only possible value of `DFTI_PACKED_FORMAT` is `DFTI_CCE_FORMAT`; this association of configuration parameters is supported for transforms of any dimension. For a description of the corresponding packed format, see [DFTI_CONJUGATE_EVEN_STORAGE](#). For one-dimensional transforms (only) with `DFTI_CONJUGATE_EVEN_STORAGE` set to `DFTI_COMPLEX_REAL`, the `DFTI_PACKED_FORMAT` configuration parameter must be `DFTI_CCS_FORMAT`, `DFTI_PACK_FORMAT`, or `DFTI_PERM_FORMAT`. The corresponding packed formats are explained and illustrated below.

DFTI_CCS_FORMAT for One-dimensional Transforms

The following figure illustrates the storage of a one-dimensional (1D) size- N conjugate-even sequence in a real array for the CCS, PACK, and PERM packed formats. The CCS format requires an array of size $N+2$, while the other formats require an array of size N . Zero-based indexing is used.

Storage of a 1D Size- N Conjugate-even Sequence in a Real Array

	n=	0	1	2	3	...	2L-2	2L-1	2L	2L+1	2L+2
CCS, N=2L		R_0	0	R_1	I_1	...	R_{L-1}	I_{L-1}	R_L	0	
PACK, N=2L		R_0	R_1	I_1	R_2	...	I_{L-1}	R_L			
PERM, N=2L		R_0	R_L	R_1	I_1	...	R_{L-1}	I_{L-1}			
CCS, N=2L+1		R_0	0	R_1	I_1	...	R_{L-1}	I_{L-1}	R_L	I_L	not used
PACK, N=2L+1		R_0	R_1	I_1	R_2	...	I_{L-1}	R_L	I_L		
PERM, N=2L+1		R_0	R_1	I_1	R_2	...	I_{L-1}	R_L	I_L		

NOTE For storage of a one-dimensional conjugate-even sequence in a real array, CCS is in the same format as CCE.

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in a real-valued array AC as illustrated by figure "Storage of a 1D Size- N Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```
float *AR; // malloc( sizeof(float)*N )
float *AC; // malloc( sizeof(float)*(N+2) )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_CCS_FORMAT );
...
// on input: R[k] = AR[k]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output:
// for k=0...N/2:   Z[k] = AC[2*k+0] + I*AC[2*k+1]
// for k=N/2+1...N-1: Z[k] = AC[2*(N-k)%N + 0] - I*AC[2*(N-k)%N + 1]
```

DFTI_PACK_FORMAT for One-dimensional Transforms

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in a real-valued array AC as illustrated by figure "Storage of a 1D Size- N Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```
float *AR; // malloc( sizeof(float)*N )
float *AC; // malloc( sizeof(float)*N )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PACK_FORMAT );
...
// on input: R[k] = AR[k]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output: Z[k] = re + I*im, where
```

```
// if (k == 0) {
//     re = AC[0];
//     im = 0;
// } else if (k == N-k) {
//     re = AC[2*k-1];
//     im = 0;
// } else if (k <= N/2) {
//     re = AC[2*k-1];
//     im = AC[2*k-0];
// } else {
//     re = AC[2*(N-k)-1];
//     im = -AC[2*(N-k)-0];
// }
```

DFTI_PERM_FORMAT for One-dimensional Transforms

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in real-valued array AC as illustrated by figure ["Storage of a 1D Size-N Conjugate-even Sequence in a Real Array"](#) and can be used to reconstruct the whole conjugate-even sequence as follows:

```
float *AR; // malloc( sizeof(float)*N )
float *AC; // malloc( sizeof(float)*N )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PERM_FORMAT );
...
// on input: R{k} = AR[k]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output: Z{k} = re + I*im, where
// if (k == 0) {
//     re = AC[0];
//     im = 0;
// } else if (k == N-k) {
//     re = AC[1];
//     im = 0;
// } else if (k <= N/2) {
//     re = AC[2*k+0 - N%2];
//     im = AC[2*k+1 - N%2];
// } else {
//     re = AC[2*(N-k)+0 - N%2];
//     im = -AC[2*(N-k)+1 - N%2];
// }
```

See Also

[DftiSetValue](#)

DFTI_WORKSPACE

The computation step for some FFT algorithms requires a scratch space for permutation or other purposes. To manage the use of the auxiliary storage, Intel® oneAPI Math Kernel Library (oneMKL) enables you to set the configuration parameter `DFTI_WORKSPACE` with the following values:

<code>DFTI_ALLOW</code>	(default) Permits the use of the auxiliary storage.
<code>DFTI_AVOID</code>	Instructs Intel® oneAPI Math Kernel Library (oneMKL) to avoid using the auxiliary storage if possible.

See Also

[DftiSetValue](#)

DFTI_COMMIT_STATUS

The `DFTI_COMMIT_STATUS` configuration parameter indicates whether the descriptor is ready for computation. The parameter has two possible values:

<code>DFTI_UNCOMMITTED</code>	Default value, set after a successful call of <code>DftiCreateDescriptor</code> .
<code>DFTI_COMMITTED</code>	The value after a successful call to <code>DftiCommitDescriptor</code> .

A computation function called with an uncommitted descriptor returns an error.

You cannot directly set this configuration parameter in a call to `DftiSetValue`, but a change in the configuration of a committed descriptor may change the commit status of the descriptor to `DFTI_UNCOMMITTED`.

See Also

[DftiCreateDescriptor](#)
[DftiCommitDescriptor](#)
[DftiSetValue](#)

DFTI_ORDERING

Some FFT algorithms apply an explicit permutation stage that is time consuming [4]. The exclusion of this step is similar to applying an FFT to input data whose order is scrambled, or allowing a scrambled order of the FFT results. In applications such as convolution and power spectrum calculation, the order of result or data is unimportant and thus using scrambled data is acceptable if it leads to better performance. The following options are available in Intel® oneAPI Math Kernel Library (oneMKL):

- `DFTI_ORDERED`: Forward transform data ordered, backward transform data ordered (default option).
- `DFTI_BACKWARD_SCRAMBLLED`: Forward transform data ordered, backward transform data scrambled.

Table "Scrambled Order Transform" tabulates the effect of this configuration setting.

Scrambled Order Transform

	<code>DftiComputeForward</code>	<code>DftiComputeBackward</code>
DFTI_ORDERING	Input → Output	Input → Output
<code>DFTI_ORDERED</code>	ordered → ordered	ordered → ordered
<code>DFTI_BACKWARD_SCRAMBLLED</code>	ordered → scrambled	scrambled → ordered

NOTE

The word "scrambled" in this table means "permit scrambled order if possible". In some situations permitting out-of-order data gives no performance advantage and an implementation may choose to ignore the suggestion.

See Also

[DftiSetValue](#)

DFTI_DESTROY_INPUT

Allowing the library to overwrite your input data in case of out-of-place transforms may be an appropriate alternative to using `DFTI_WORKSPACE` to reduce the memory footprint of your application. Intel® oneAPI Math Kernel Library (oneMKL) enables you to communicate whether this is allowed via the configuration parameter `DFTI_DESTROY_INPUT`, with the following values:

<code>DFTI_AVOID</code>	The default. Instructs the library to leave the input data unchanged.
-------------------------	---

DFTI_ALLOW

Permits the input data to be overwritten.

See Also

[DftiSetValue](#)

FFT Descriptor Manipulation Functions

This category contains the following functions: create a descriptor, commit a descriptor, copy a descriptor, and free a descriptor.

DftiCreateDescriptor

Allocates the descriptor data structure and initializes it with default configuration values.

Syntax

```
status = DftiCreateDescriptor(&desc_handle, precision, forward_domain, dimension,
length);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>precision</i>	enum	Precision of the transform: DFTI_SINGLE or DFTI_DOUBLE.
<i>forward_domain</i>	enum	Forward domain of the transform: DFTI_COMPLEX or DFTI_REAL.
<i>dimension</i>	MKL_LONG	Dimension of the transform.
<i>length</i>	MKL_LONG if <i>dimension</i> = 1. Array of type MKL_LONG otherwise.	Length of the transform for a one-dimensional transform. Lengths of each dimension for a multi-dimensional transform.

Output Parameters

Name	Type	Description
<i>desc_handle</i>	DFTI_DESCRIPTOR_HANDLE	FFT descriptor.
<i>status</i>	MKL_LONG	Function completion status.

Description

This function allocates memory for the descriptor data structure and instantiates it with all the default configuration settings for the precision, forward domain, dimension, and length of the desired transform. Because memory is allocated dynamically, the result is actually a pointer to the created descriptor. This function is slightly different from the "initialization" function that can be found in software packages or libraries that implement more traditional algorithms for computing an FFT. This function does not perform any significant computational work such as computation of twiddle factors. The function [DftiCommitDescriptor](#) does this work after the function [DftiSetValue](#) has set values of all necessary parameters.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Prototype

```
/* Note that the preprocessor definition provided below only illustrates
 * that the actual function called may be determined at compile time.
 * You can rely only on the declaration of the function.
 * For precise definition of the preprocessor macro, see the include/mkl_dfti.h
 * file in the Intel MKL directory.
 */
MKL_LONG DftiCreateDescriptor(DFTI_DESCRIPTOR_HANDLE * pHandle,
    enum DFTI_CONFIG_VALUE precision,
    enum DFTI_CONFIG_VALUE domain,
    MKL_LONG dimension, ... /* length/lengths */ );

#define DftiCreateDescriptor(desc, prec, domain, dim, sizes) \
    ((prec)==DFTI_SINGLE && (dim)==1) ? \
    some_actual_function_sld((desc), (domain), (MKL_LONG)(sizes)) : \
    ...
```

Variable *length/lengths* is interpreted as a scalar (MKL_LONG) or an array (MKL_LONG*), depending on the value of parameter *dimension*. If the value of parameter *precision* is known at compile time, an optimizing compiler retains only the call to the respective specific function, thereby reducing the size of the statically linked application. Avoid direct calls to the specific functions used in the preprocessor macro definition, because their interface may change in future releases of the library. If the use of the macro is undesirable, you can safely undefine it after inclusion of the Intel® oneAPI Math Kernel Library (oneMKL) FFT header file, as follows:

```
#include "mkl_dfti.h"
#undef DftiCreateDescriptor
```

See Also

[DFTI_PRECISION](#) configuration parameter

[DFTI_FORWARD_DOMAIN](#) configuration parameter

[DFTI_DIMENSION](#), [DFTI_LENGTHS](#) configuration parameters

[Configuration Parameters, summary table](#)

DftiCommitDescriptor

Performs all initialization for the actual FFT computation.

Syntax

```
status = DftiCommitDescriptor(desc_handle);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>desc_handle</i>	DFTI_DESCRIPTOR_HANDLE	FFT descriptor.

Output Parameters

Name	Type	Description
<code>desc_handle</code>	<code>DFTI_DESCRIPTOR_HANDLE</code>	Updated FFT descriptor.
<code>status</code>	<code>MKL_LONG</code>	Function completion status.

Description

This function completes initialization of a previously created descriptor, which is required before the descriptor can be used for FFT computations. Typically, committing the descriptor performs all initialization that is required for the actual FFT computation. The initialization done by the function may involve exploring different factorizations of the input length to find the optimal computation method.

If you call the `DftiSetValue` function to change configuration parameters of a committed descriptor (see [Descriptor Configuration Functions](#)), you must re-commit the descriptor before invoking a computation function. Typically, a commitment function call is immediately followed by a computation function call (see [FFT Computation Functions](#)).

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Prototype

```
MKL_LONG DftiCommitDescriptor( DFTI_DESCRIPTOR_HANDLE );
```

DftiFreeDescriptor

Frees the memory allocated for a descriptor.

Syntax

```
status = DftiFreeDescriptor(&desc_handle);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>desc_handle</code>	<code>DFTI_DESCRIPTOR_HANDLE</code>	FFT descriptor.

Output Parameters

Name	Type	Description
<code>desc_handle</code>	<code>DFTI_DESCRIPTOR_HANDLE</code>	Memory for the FFT descriptor is released.
<code>status</code>	<code>MKL_LONG</code>	Function completion status.

Description

This function frees all memory allocated for a descriptor.

NOTE

Memory allocation/deallocation inside Intel® oneAPI Math Kernel Library (oneMKL) is managed by Intel® oneAPI Math Kernel Library (oneMKL) memory management software. So, even after successful completion of `FreeDescriptor`, the memory space may continue being allocated for the application because the memory management software sometimes does not return the memory space to the OS, but considers the space free and can reuse it for future memory allocation. See [Example mkl_free_buffers: Usage with FFT Functions](#) on how to use Intel® oneAPI Math Kernel Library (oneMKL) memory management software and release memory to the OS.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Prototype

```
MKL_LONG DftiFreeDescriptor( DFTI_DESCRIPTOR_HANDLE * );
```

DftiCopyDescriptor

Makes a copy of an existing descriptor.

Syntax

```
status = DftiCopyDescriptor(desc_handle_original, &desc_handle_copy);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>desc_handle_original</code>	<code>DFTI_DESCRIPTOR_HANDLE</code>	The FFT descriptor to copy.
	<code>DFTI_DESCRIPTOR_HANDLE</code>	The FFT descriptor to copy.

Output Parameters

Name	Type	Description
<code>dfti_handle_dst</code>	<code>DFTI_DESCRIPTOR_HANDLE</code>	The copy of the FFT descriptor.
<code>status</code>	<code>MKL_LONG</code>	Function completion status.

Description

This function makes a copy of an existing descriptor. The resulting descriptor `desc_handle_copy` and the existing descriptor `desc_handle_original` specify the same configuration of the transform, but do not have any memory areas in common ("deep copy").

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Prototype

```
MKL_LONG DftiCopyDescriptor( DFTI_DESCRIPTOR_HANDLE, DFTI_DESCRIPTOR_HANDLE * );
```

FFT Descriptor Configuration Functions

This category contains the following functions: the value setting function [DftiSetValue](#) sets one particular configuration parameter to an appropriate value, and the value-getting function [DftiGetValue](#) reads the value of one particular configuration parameter. While all configuration parameters are readable, you cannot set a few of them. Some of these contain fixed information of a particular implementation such as version number, or dynamic information, which is derived by the implementation during execution of one of the functions. See [Configuration Settings](#) for details.

DftiSetValue

Sets one particular configuration parameter with the specified configuration value.

Syntax

```
status = DftiSetValue(desc_handle, config_param, config_val);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>desc_handle</i>	DFTI_DESCRIPTOR_HANDLE	FFT descriptor.
<i>config_param</i>	enum	Configuration parameter.
<i>config_val</i>	Depends on the configuration parameter.	Configuration value.

Output Parameters

Name	Type	Description
<i>desc_handle</i>	DFTI_DESCRIPTOR_HANDLE	Updated FFT descriptor.
<i>status</i>	MKL_LONG	Function completion status.

Description

This function sets one particular configuration parameter with the specified configuration value. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see:

- [DFTI_PRECISION](#)
- [DFTI_FORWARD_DOMAIN](#)
- [DFTI_DIMENSION](#), [DFTI_LENGTHS](#)
- [DFTI_PLACEMENT](#)
- [DFTI_FORWARD_SCALE](#), [DFTI_BACKWARD_SCALE](#)

- [DFTI_THREAD_LIMIT](#)
- [DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES](#)
- [DFTI_NUMBER_OF_TRANSFORMS](#)
- [DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE](#)
- [DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE](#)
- [DFTI_PACKED_FORMAT](#)
- [DFTI_WORKSPACE](#)
- [DFTI_ORDERING](#)
- [DFTI_DESTROY_INPUT](#)

You cannot use the `DftiSetValue` function to change configuration parameters `DFTI_FORWARD_DOMAIN`, `DFTI_PRECISION`, `DFTI_DIMENSION`, and `DFTI_LENGTHS`. Use the `DftiCreateDescriptor` function to set them.

Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in C C++](#).

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Prototype

```
MKL_LONG DftiSetValue( DFTI_DESCRIPTOR_HANDLE, DFTI_CONFIG_PARAM , ... );
```

See Also

[Configuration Settings](#) for more information on configuration parameters.

[DftiCreateDescriptor](#)

[DftiGetValue](#)

DftiGetValue

Gets the configuration value of one particular configuration parameter.

Syntax

```
status = DftiGetValue(desc_handle, config_param, &config_val);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>desc_handle</code>	<code>DFTI_DESCRIPTOR_HANDLE</code>	FFT descriptor.
<code>config_param</code>	enum	Configuration parameter. See Table "Configuration Parameters" for allowable values of <code>config_param</code> .

Output Parameters

Name	Type	Description
<code>config_val</code>	Depends on the configuration parameter.	Configuration value.
<code>status</code>	MKL_LONG	Function completion status.

Description

This function gets the configuration value of one particular configuration parameter. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see:

- `DFTI_PRECISION`
- `DFTI_FORWARD_DOMAIN`
- `DFTI_DIMENSION`, `DFTI_LENGTH`
- `DFTI_PLACEMENT`
- `DFTI_FORWARD_SCALE`, `DFTI_BACKWARD_SCALE`
- `DFTI_THREAD_LIMIT`
- `DFTI_INPUT_STRIDES`, `DFTI_OUTPUT_STRIDES`
- `DFTI_NUMBER_OF_TRANSFORMS`
- `DFTI_INPUT_DISTANCE`, `DFTI_OUTPUT_DISTANCE`
- `DFTI_COMPLEX_STORAGE`, `DFTI_REAL_STORAGE`, `DFTI_CONJUGATE_EVEN_STORAGE`
- `DFTI_PACKED_FORMAT`
- `DFTI_WORKSPACE`
- `DFTI_COMMIT_STATUS`
- `DFTI_ORDERING`
- `DFTI_DESTROY_INPUT`

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Prototype

```
MKL_LONG DftiGetValue( DFTI_DESCRIPTOR_HANDLE,
DFTI_CONFIG_PARAM ,
... );
```

[Configuration Settings](#) for more information on configuration parameters.

`DftiSetValue`

FFT Computation Functions

This category contains the following functions: compute the forward transform and compute the backward transform.

DftiComputeForward

Computes the forward FFT.

Syntax

```
status = DftiComputeForward(desc_handle, x_inout);
```

```
status = DftiComputeForward(desc_handle, x_in, y_out);
status = DftiComputeForward(desc_handle, xre_inout, xim_inout);
status = DftiComputeForward(desc_handle, xre_in, xim_in, yre_out, yim_out);
```

Input Parameters

Name	Type	Description
<code>desc_handle</code>	<code>DFTI_DESCRIPTOR_HANDLE</code>	FFT descriptor.
<code>x_inout, x_in</code>	Array of type <code>float</code> or <code>double</code> depending on the precision of the transform.	Data to be transformed in case of a real forward domain or, in the case of a complex forward domain in association with <code>DFTI_COMPLEX_COMPLEX</code> , set for <code>DFTI_COMPLEX_STORAGE</code> .
<code>xre_inout, xim_inout, xre_in, xim_in</code>	Array of type <code>float</code> or <code>double</code> depending on the precision of the transform.	Real and imaginary parts of the data to be transformed in the case of a complex forward domain.

The suffix in parameter names corresponds to the value of the configuration parameter `DFTI_PLACEMENT` as follows:

- `_inout` to `DFTI_INPLACE`
- `_in` to `DFTI_NOT_INPLACE`

Output Parameters

Name	Type	Description
<code>y_out</code>	Array of type <code>float</code> or <code>double</code> depending on the precision of the transform.	The transformed data in case of a real backward domain or, in the case of a complex forward domain in association with <code>DFTI_COMPLEX_COMPLEX</code> , set for <code>DFTI_COMPLEX_STORAGE</code> .
<code>xre_inout, xim_inout, yre_out, yim_out</code>	Array of type <code>float</code> or <code>double</code> depending on the precision of the transform.	Real and imaginary parts of the transformed data in the case of a complex forward domain in association with <code>DFTI_REAL_REAL</code> set for <code>DFTI_COMPLEX_STORAGE</code> .
<code>status</code>	<code>MKL_LONG</code>	Function completion status.

The suffix in parameter names corresponds to the value of the configuration parameter `DFTI_PLACEMENT` as follows:

- `_inout` to `DFTI_INPLACE`
- `_out` to `DFTI_NOT_INPLACE`

Include Files

- `mkl.h`

Description

The `DftiComputeForward` function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, this function computes the forward FFT, that is, the [transform](#) with the minus sign in the exponent, $\delta = -1$.

The `DFTI_COMPLEX_STORAGE`, `DFTI_REAL_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the layout of the input and output data and must be properly set in a call to the `DftiSetValue` function. The forward domain and the precision of the transform are determined by the configuration settings `DFTI_FORWARD_DOMAIN` and `DFTI_PRECISION`, which are during construction of the descriptor.

The FFT descriptor must be properly configured prior to the function call. Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in C C++](#).

The number and types of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by variable parameters.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Prototype

```
MKL_LONG DftiComputeForward( DFTI_DESCRIPTOR_HANDLE, void*, ... );
```

See Also

[Configuration Settings](#)

[DFTI_FORWARD_DOMAIN](#)

[DFTI_PLACEMENT](#)

[DFTI_PACKED_FORMAT](#)

[DFTI_COMPLEX_STORAGE](#), [DFTI_REAL_STORAGE](#), [DFTI_CONJUGATE_EVEN_STORAGE](#)

[DFTI_DIMENSION](#), [DFTI_LENGTHS](#)

[DFTI_INPUT_DISTANCE](#), [DFTI_OUTPUT_DISTANCE](#)

[DFTI_INPUT_STRIDES](#), [DFTI_OUTPUT_STRIDES](#)

[DftiComputeBackward](#)

[DftiSetValue](#)

DftiComputeBackward

Computes the backward FFT.

Syntax

```
status = DftiComputeBackward(desc_handle, x_inout);
```

```
status = DftiComputeBackward(desc_handle, y_in, x_out);
```

```
status = DftiComputeBackward(desc_handle, xre_inout, xim_inout);
```

```
status = DftiComputeBackward(desc_handle, yre_in, yim_in, xre_out, xim_out);
```

Input Parameters

Name	Type	Description
<code>desc_handle</code>	<code>DFTI_DESCRIPTOR_HANDLE</code>	FFT descriptor.

Name	Type	Description
<code>x_inout, y_in</code>	Array of type <code>float</code> or <code>double</code> depending on the precision of the transform.	Data to be transformed in case of a real forward domain or, in the case of a complex forward domain in association with <code>DFTI_COMPLEX_COMPLEX</code> , set for <code>DFTI_COMPLEX_STORAGE</code> .
<code>xre_inout, xim_inout, yre_in, yim_in</code>	Array of type <code>float</code> or <code>double</code> depending on the precision of the transform.	Real and imaginary parts of the data to be transformed in the case of a complex forward domain in association with <code>DFTI_REAL_REAL</code> set for <code>DFTI_COMPLEX_STORAGE</code> .

The suffix in parameter names corresponds to the value of the configuration parameter `DFTI_PLACEMENT` as follows:

- `_inout` to `DFTI_INPLACE`
- `_in` to `DFTI_NOT_INPLACE`

Output Parameters

Name	Type	Description
<code>x_out</code>	Array of type <code>float</code> or <code>double</code> depending on the precision of the transform.	The transformed data in case of a real forward domain or, in the case of a complex forward domain in association with <code>DFTI_COMPLEX_COMPLEX</code> , set for <code>DFTI_COMPLEX_STORAGE</code> .
<code>xre_inout, xim_inout, xre_out, xim_out</code>	Array of type <code>float</code> or <code>double</code> depending on the precision of the transform.	Real and imaginary parts of the transformed data in the case of a complex forward domain in association with <code>DFTI_REAL_REAL</code> set for <code>DFTI_COMPLEX_STORAGE</code> .
<code>status</code>	<code>MKL_LONG</code>	Function completion status.

The suffix in parameter names corresponds to the value of the configuration parameter `DFTI_PLACEMENT` as follows:

- `_inout` to `DFTI_INPLACE`
- `_out` to `DFTI_NOT_INPLACE`

Include Files

- `mkl.h`

Description

The function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, the `DftiComputeBackward` function computes the inverse FFT, that is, the [transform](#) with the plus sign in the exponent, $\delta = +1$.

The `DFTI_COMPLEX_STORAGE`, `DFTI_REAL_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the layout of the input and output data and must be properly set in a call to the `DftiSetValue` function. The forward domain and the precision of the transform are determined by the configuration settings `DFTI_FORWARD_DOMAIN` and `DFTI_PRECISION`, which are during construction of the descriptor.

The FFT descriptor must be properly configured prior to the function call. Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in C C++](#).

The number and types of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by variable parameters.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Prototype

```
MKL_LONG DftiComputeBackward( DFTI_DESCRIPTOR_HANDLE, void *, ... );
```

See Also

Configuration Settings

`DFTI_FORWARD_DOMAIN`

`DFTI_PLACEMENT`

`DFTI_PACKED_FORMAT`

`DFTI_COMPLEX_STORAGE`, `DFTI_REAL_STORAGE`, `DFTI_CONJUGATE_EVEN_STORAGE`

`DFTI_DIMENSION`, `DFTI_LENGTHS`

`DFTI_INPUT_DISTANCE`, `DFTI_OUTPUT_DISTANCE`

`DFTI_INPUT_STRIDES`, `DFTI_OUTPUT_STRIDES`

`DftiComputeForward`

`DftiSetValue`

Configuring and Computing an FFT in C/C++

The table below summarizes information on configuring and computing an FFT in C/C++ for all kinds of transforms and possible combinations of input and output domains.

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
Complex-to-complex, in-place, forward or backward	Interleaved complex numbers	Interleaved complex numbers	<pre>/* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_COMPLEX, <dimension>, <sizes>); status = DftiCommitDescriptor(hand); /* Compute an FFT */ /* forward FFT */ status = DftiComputeForward(hand, X_inout); /* or backward FFT */ status = DftiComputeBackward(hand, X_inout);</pre>
Complex-to-complex, out-of-place,	Interleaved complex numbers	Interleaved complex numbers	<pre>/* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_COMPLEX, <dimension>,</pre>

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
forward or backward			<pre> <sizes>; status = DftiSetValue(hand, DFTI_PLACEMENT, DFTI_NOT_INPLACE); status = DftiCommitDescriptor(hand); /* Compute an FFT */ /* forward FFT */ status = DftiComputeForward(hand, X_in, Y_out); /* or backward FFT */ status = DftiComputeBackward(hand, X_in, Y_out); </pre>
Complex-to-complex, in-place, forward or backward	Split-complex numbers	Split-complex numbers	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_COMPLEX, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_COMPLEX_STORAGE, DFTI_REAL_REAL); status = DftiCommitDescriptor(hand); /* Compute an FFT */ /* forward FFT */ status = DftiComputeForward(hand, Xre_inout, Xim_inout); /* or backward FFT */ status = DftiComputeBackward(hand, Xre_inout, Xim_inout); </pre>
Complex-to-complex, out-of-place, forward or backward	Split-complex numbers	Split-complex numbers	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_COMPLEX, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_COMPLEX_STORAGE, DFTI_REAL_REAL); status = DftiSetValue(hand, DFTI_PLACEMENT, DFTI_NOT_INPLACE); status = DftiCommitDescriptor(hand); /* Compute an FFT */ /* forward FFT */ status = DftiComputeForward(hand, Xre_in, Xim_in, Yre_out, Yim_out); /* or backward FFT */ status = DftiComputeBackward(hand, Xre_in, Xim_in, Yre_out, Yim_out); </pre>
Real-to-complex, in-place, forward	Real numbers	Numbers in the CCE format	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_REAL, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX); status = DftiSetValue(hand, </pre>

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
			<pre> DFTI_PACKED_FORMAT, DFTI_CCE_FORMAT); status = DftiSetValue(hand, DFTI_INPUT_STRIDES, <real_strides>); status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, <complex_strides>); status = DftiCommitDescriptor(hand); /* Compute an FFT */ status = DftiComputeForward(hand, X_inout); </pre>
Real-to-complex, out-of-place, forward	Real numbers	Numbers in the CCE format	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_REAL, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX); status = DftiSetValue(hand, DFTI_PACKED_FORMAT, DFTI_CCE_FORMAT); status = DftiSetValue(hand, DFTI_PLACEMENT, DFTI_NOT_INPLACE); status = DftiSetValue(hand, DFTI_INPUT_STRIDES, <real_strides>); status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, <complex_strides>); status = DftiCommitDescriptor(hand); /* Compute an FFT */ status = DftiComputeForward(hand, X_in, Y_out); </pre>
Complex-to-real, in-place, backward	Numbers in the CCE format	Real numbers	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_REAL, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX); status = DftiSetValue(hand, DFTI_PACKED_FORMAT, DFTI_CCE_FORMAT); status = DftiSetValue(hand, DFTI_INPUT_STRIDES, <complex_strides>); status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, <real_strides>); status = DftiCommitDescriptor(hand); /* Compute an FFT */ status = DftiComputeBackward(hand, X_inout); </pre>
Complex-to-real, out-of-place, backward	Numbers in the CCE format	Real numbers	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_REAL, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX); </pre>

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
			<pre> status = DftiSetValue(hand, DFTI_PLACEMENT, DFTI_NOT_INPLACE); status = DftiSetValue(hand, DFTI_PACKED_FORMAT, DFTI_CCE_FORMAT); status = DftiSetValue(hand, DFTI_INPUT_STRIDES, <complex_strides>); status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, <real_strides>); status = DftiCommitDescriptor(hand); /* Compute an FFT */ status = DftiComputeBackward(hand, X_in, Y_out); </pre>

You can find C programs that illustrate configuring and computing FFTs in the `examples/dftc/` subdirectory of your Intel® oneAPI Math Kernel Library (oneMKL) directory.

Status Checking Functions

All of the descriptor manipulation, FFT computation, and descriptor configuration functions return an integer value denoting the status of the operation. The functions in this category check that status. The first function is a logical function that checks whether the status reflects an error of a predefined class, and the second is an error message function that returns a character string.

DftiErrorClass

Checks whether the status reflects an error of a predefined class.

Syntax

```
predicate = DftiErrorClass(status, error_class);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>status</code>	<code>MKL_LONG</code>	Completion status of a fast Fourier transform (FFT) function.
<code>error_class</code>	<code>MKL_LONG</code>	Predefined error class.

Output Parameters

Name	Type	Description
<code>predicate</code>	<code>MKL_LONG</code>	Result of checking.

Description

The FFT interface in Intel® oneAPI Math Kernel Library (oneMKL) provides a set of predefined error classes listed in [Table "Predefined Error Classes"](#). They are named constants and have the type `MKL_LONG`.

Predefined Error Classes

Named Constants	Comments
DFTI_NO_ERROR	No error. The zero status belongs to this class.
DFTI_MEMORY_ERROR	Usually associated with memory allocation.
DFTI_INVALID_CONFIGURATION	Invalid settings in one or more configuration parameters.
DFTI_INCONSISTENT_CONFIGURATION	Inconsistent configuration or input parameters.
DFTI_NUMBER_OF_THREADS_ERROR	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function).
DFTI_MULTITHREADED_ERROR	Usually associated with a value that OMP routines return in case of errors.
DFTI_BAD_DESCRIPTOR	Descriptor is unusable for computation.
DFTI_UNIMPLEMENTED	Unimplemented legitimate settings; implementation dependent.
DFTI_MKL_INTERNAL_ERROR	Internal library error.
DFTI_1D_LENGTH_EXCEEDS_INT32	Length of one of the dimensions exceeds $2^{32} - 1$ (4 bytes).
DFTI_1D_MEMORY_EXCEEDS_INT32	Data size of one of the transforms exceeds $2^{31} - 1$ bytes.

NOTE

Use `DFTI_1D_MEMORY_EXCEEDS_INT32` instead of `DFTI_1D_LENGTH_EXCEEDS_INT32` for better accuracy.

The `DftiErrorClass` function returns a non-zero value if the status belongs to the predefined error class. To check whether a function call was successful, call `DftiErrorClass` with a specific error class. However, the zero value of the status belongs to the `DFTI_NO_ERROR` class and thus the zero status indicates successful completion of an operation. See [Example "Using Status Checking Functions"](#) for an illustration of correct use of the status checking functions.

NOTE

It is incorrect to directly compare a status with a predefined class.

Prototype

```
MKL_LONG DftiErrorClass( MKL_LONG , MKL_LONG );
```

DftiErrorMessage

Generates an error message.

Syntax

```
error_message = DftiErrorMessage(status);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>status</i>	<code>MKL_LONG</code>	Completion status of a function.

Output Parameters

Name	Type	Description
<i>error_message</i>	Array of <code>char</code>	The character string with the error message.

Description

The error message function generates an error message character string. The function returns a pointer to a constant character string, that is, a character array with terminating '\0' character, and you do not need to free this pointer.

[Example Using Status Checking Function](#) shows how this function can be used.

Prototype

```
char *DftiErrorMessage( MKL_LONG );
```

Cluster FFT Functions

This section describes the cluster Fast Fourier Transform (FFT) functions implemented in Intel® oneAPI Math Kernel Library (oneMKL).

NOTE

These functions are available only for Intel® 64 architectures.

The cluster FFT function library was designed to perform fast Fourier transforms on a cluster, that is, a group of computers interconnected via a network. Each computer (node) in the cluster has its own memory and processor(s). Data interchanges between the nodes are provided by the network.

One or more processes may be running in parallel on each cluster node. To organize communication between different processes, the cluster FFT function library uses the Message Passing Interface (MPI). To avoid dependence on a specific MPI implementation (for example, MPICH, Intel® MPI, and others), the library works with MPI via a message-passing library for linear algebra called BLACS.

Cluster FFT functions of Intel® oneAPI Math Kernel Library (oneMKL) provide one-dimensional, two-dimensional, and multi-dimensional (up to the order of 7) functions and both Fortran and C interfaces for all transform functions.

To develop applications using the cluster FFT functions, you should have basic skills in MPI programming.

The interfaces for the Intel® oneAPI Math Kernel Library (oneMKL) cluster FFT functions are similar to the corresponding interfaces for the conventional Intel® oneAPI Math Kernel Library (oneMKL) [FFT functions](#). Refer there for details not explained in this section.

Table "Cluster FFT Functions in Intel® oneAPI Math Kernel Library (oneMKL)" lists cluster FFT functions implemented in Intel® oneAPI Math Kernel Library (oneMKL):

Cluster FFT Functions in oneMKL

Function Name	Operation
Descriptor Manipulation Functions	
DftiCreateDescriptorDM	Allocates memory for the descriptor data structure and preliminarily initializes it.
DftiCommitDescriptorDM	Performs all initialization for the actual FFT computation.
DftiFreeDescriptorDM	Frees memory allocated for a descriptor.
FFT Computation Functions	
DftiComputeForwardDM	Computes the forward FFT.
DftiComputeBackwardDM	Computes the backward FFT.
Descriptor Configuration Functions	
DftiSetValueDM	Sets one particular configuration parameter with the specified configuration value.
DftiGetValueDM	Gets the value of one particular configuration parameter.

Computing Cluster FFT

The Intel® oneAPI Math Kernel Library (oneMKL) cluster FFT functions are provided with Fortran and C interfaces.

Cluster FFT computation is performed by [DftiComputeForwardDM](#) and [DftiComputeBackwardDM](#) functions, called in a program using MPI, which will be referred to as MPI program. After an MPI program starts, a number of processes are created. MPI identifies each process by its rank. The processes are independent of one another and communicate via MPI. A function called in an MPI program is invoked in all the processes. Each process manipulates data according to its rank. Input or output data for a cluster FFT transform is a sequence of real or complex values. A cluster FFT computation function operates on the local part of the input data, that is, some part of the data to be operated in a particular process, as well as generates local part of the output data. While each process performs its part of computations, running in parallel and communicating through MPI, the processes perform the entire FFT computation. FFT computations using the Intel® oneAPI Math Kernel Library (oneMKL) cluster FFT functions are typically effected by a number of steps listed below:

1. Initiate MPI by calling `MPI_Init` (the function must be called prior to calling any FFT function and any MPI function).
2. Allocate memory for the descriptor and create it by calling [DftiCreateDescriptorDM](#).
3. Specify one of several values of configuration parameters by one or more calls to [DftiSetValueDM](#).
4. Obtain values of configuration parameters needed to create local data arrays; the values are retrieved by calling [DftiGetValueDM](#).
5. Initialize the descriptor for the FFT computation by calling [DftiCommitDescriptorDM](#).
6. Create arrays for local parts of input and output data and fill the local part of input data with values. (For more information, see [Distributing Data among Processes](#).)
7. Compute the transform by calling [DftiComputeForwardDM](#) or [DftiComputeBackwardDM](#).
8. Gather local output data into the global array using MPI functions. (This step is optional because you may need to immediately employ the data differently.)
9. Release memory allocated for the descriptor by calling [DftiFreeDescriptorDM](#).
10. Finalize communication through MPI by calling `MPI_Finalize` (the function must be called after the last call to a cluster FFT function and the last call to an MPI function).

Several code examples in [Examples for Cluster FFT Functions](#) in the Code Examples appendix illustrate cluster FFT computations.

Distributing Data Among Processes

The Intel® oneAPI Math Kernel Library (oneMKL) cluster FFT functions store all input and output multi-dimensional arrays (matrices) in one-dimensional arrays (vectors). The arrays are stored in the row-major order. For example, a two-dimensional matrix A of size (m,n) is stored in a vector B of size $m*n$ so that

$$B[i*n+j]=A[i][j] \quad (i=0, \dots, m-1, j=0, \dots, n-1).$$

NOTE

Order of FFT dimensions is the same as the order of array dimensions in the programming language. For example, a 3-dimensional FFT with Lengths= (m,n,l) can be computed over an array $Ar[m][n][l]$.

All MPI processes involved in cluster FFT computation operate their own portions of data. These local arrays make up the virtual global array that the fast Fourier transform is applied to. It is your responsibility to properly allocate local arrays (if needed), fill them with initial data and gather resulting data into an actual global array or process the resulting data differently. To be able to do this, see sections below on how the virtual global array is composed of the local ones.

Multi-dimensional transforms

If the dimension of transform is greater than one, the cluster FFT function library splits data in the dimension whose index changes most slowly, so that the parts contain all elements with several consecutive values of this index. It is the first dimension in C . If the global array is two-dimensional, it gives each process several consecutive rows. Local arrays are placed in memory allocated for the virtual global array consecutively, in the order determined by process ranks. For example, in case of two processes, during the computation of a three-dimensional transform whose matrix has size $(11,15,12)$, the processes may store local arrays of sizes $(6,15,12)$ and $(5,15,12)$, respectively.

If p is the number of MPI processes and the matrix of a transform to be computed has size (m,n,l) , each MPI process works with local data array of size (m_q, n, l) , where $\sum m_q = m$, $q=0, \dots, p-1$. Local input arrays must contain appropriate parts of the actual global input array, and then local output arrays will contain appropriate parts of the actual global output array. You can figure out which particular rows of the global array the local array must contain from the following configuration parameters of the cluster FFT interface: `CDFT_LOCAL_NX`, `CDFT_LOCAL_START_X`, and `CDFT_LOCAL_SIZE`. To retrieve values of the parameters, use the `DftiGetValueDM` function:

- `CDFT_LOCAL_NX` specifies how many rows of the global array the current process receives.
- `CDFT_LOCAL_START_X` specifies which row of the global input or output array corresponds to the first row of the local input or output array. If A is a global array and L is the appropriate local array, then

$$L[i][j][k]=A[i+cdft_local_start_x][j][k], \text{ where } i=0, \dots, m_q-1, j=0, \dots, n-1, k=0, \dots, l-1.$$

Example "2D Out-of-place Cluster FFT Computation" shows how the data is distributed among processes for a two-dimensional cluster FFT computation.

One-dimensional transforms

In this case, input and output data are distributed among processes differently and even the numbers of elements stored in a particular process before and after the transform may be different. Each local array stores a segment of consecutive elements of the appropriate global array. Such segment is determined by the number of elements and a shift with respect to the first array element. So, to specify segments of the global input and output arrays that a particular process receives, *four* configuration parameters are needed: `CDFT_LOCAL_NX`, `CDFT_LOCAL_START_X`, `CDFT_LOCAL_OUT_NX`, and `CDFT_LOCAL_OUT_START_X`. Use the `DftiGetValueDM` function to retrieve their values. The meaning of the four configuration parameters depends upon the type of the transform, as shown in Table "Data Distribution Configuration Parameters for 1D Transforms":

Data Distribution Configuration Parameters for 1D Transforms

Meaning of the Parameter	Forward Transform	Backward Transform
Number of elements in input array	CDFT_LOCAL_NX	CDFT_LOCAL_OUT_NX
Elements shift in input array	CDFT_LOCAL_START_X	CDFT_LOCAL_OUT_START_X
Number of elements in output array	CDFT_LOCAL_OUT_NX	CDFT_LOCAL_NX
Elements shift in output array	CDFT_LOCAL_OUT_START_X	CDFT_LOCAL_START_X

Memory size for local data

The memory size needed for local arrays cannot be just calculated from `CDFT_LOCAL_NX` (`CDFT_LOCAL_OUT_NX`), because the cluster FFT functions sometimes require allocating a little bit more memory for local data than just the size of the appropriate sub-array. The configuration parameter `CDFT_LOCAL_SIZE` specifies the size of the local input and output array in data elements. Each local input and output arrays must have size not less than `CDFT_LOCAL_SIZE*size_of_element`. Note that in the current implementation of the cluster FFT interface, data elements can be real or complex values, each complex value consisting of the real and imaginary parts. If you employ a user-defined workspace for in-place transforms (for more information, refer to [Table "Settable configuration Parameters"](#)), it must have the same size as the local arrays. [Example "1D In-place Cluster FFT Computations"](#) illustrates how the cluster FFT functions distribute data among processes in case of a one-dimensional FFT computation performed with a user-defined workspace.

Available Auxiliary Functions

If a global input array is located on one MPI process and you want to obtain its local parts or you want to gather the global output array on one MPI process, you can use functions `MKL_CDFT_ScatterData` and `MKL_CDFT_GatherData` to distribute or gather data among processes, respectively. These functions are defined in a file that is delivered with Intel® oneAPI Math Kernel Library (oneMKL) and located in the following subdirectory of the Intel® oneAPI Math Kernel Library (oneMKL) installation directory: `examples/cdftc/source/cdft_example_support.c`.

Restriction on Lengths of Transforms

The algorithm that the Intel® oneAPI Math Kernel Library (oneMKL) cluster FFT functions use to distribute data among processes imposes a restriction on lengths of transforms with respect to the number of MPI processes used for the FFT computation:

- For a multi-dimensional transform, the lengths of the first two dimensions must be not less than the number of MPI processes.
- The length of a one-dimensional transform must be the product of two integers each of which is not less than the number of MPI processes.

Non-compliance with the restriction causes an error `CDFT_SPREAD_ERROR` (refer to [Error Codes](#) for details). To achieve the compliance, you can change the transform lengths and/or the number of MPI processes, which is specified at start of an MPI program. MPI-2 enables changing the number of processes during execution of an MPI program.

Cluster FFT Interface

To use the cluster FFT functions, you need to access the header file `mkl_cdft.h` through `"include"`.

The C interface provides a structure type `DFTI_DESCRIPTOR_DM_HANDLE` and a number of functions, some of which accept a different number of input arguments.

To provide communication between parallel processes through MPI, the following include statement must be present in your code:

- C/C++:

```
#include "mpi.h"
```

There are three main categories of the cluster FFT functions in Intel® oneAPI Math Kernel Library (oneMKL):

1. **Descriptor Manipulation.** There are three functions in this category. The [DftiCreateDescriptorDM](#) function creates an FFT descriptor whose storage is allocated dynamically. The [DftiCommitDescriptorDM](#) function "commits" the descriptor to all its settings. The [DftiFreeDescriptorDM](#) function frees up the memory allocated for the descriptor.
2. **FFT Computation.** There are two functions in this category. The [DftiComputeForwardDM](#) function performs the forward FFT computation, and the [DftiComputeBackwardDM](#) function performs the backward FFT computation.
3. **Descriptor Configuration.** There are two functions in this category. The [DftiSetValueDM](#) function sets one specific configuration value to one of the many configuration parameters. The [DftiGetValueDM](#) function gets the current value of any of these configuration parameters, all of which are readable. These parameters, though many, are handled one at a time.

Cluster FFT Descriptor Manipulation Functions

There are three functions in this category: create a descriptor, commit a descriptor, and free a descriptor.

DftiCreateDescriptorDM

Allocates memory for the descriptor data structure and preliminarily initializes it.

Syntax

```
status = DftiCreateDescriptorDM(comm, &handle, v1, v2, dim, size );
status = DftiCreateDescriptorDM(comm, &handle, v1, v2, dim, sizes );
```

Include Files

- mkl_cdft.h

Input Parameters

<i>comm</i>	MPI communicator, e.g. MPI_COMM_WORLD.
<i>v1</i>	Precision of the transform.
<i>v2</i>	Type of the forward domain. Must be <code>DFTI_COMPLEX</code> for complex-to-complex transforms or <code>DFTI_REAL</code> for real-to-complex transforms.
<i>dim</i>	Dimension of the transform.
<i>size</i>	Length of the transform in a one-dimensional case.
<i>sizes</i>	Lengths of the transform in a multi-dimensional case.

Output Parameters

<i>handle</i>	Pointer to the descriptor handle of transform. If the function completes successfully, the pointer to the created handle is stored in the variable.
---------------	---

Description

This function allocates memory in a particular MPI process for the descriptor data structure and instantiates it with default configuration settings with respect to the precision, domain, dimension, and length of the desired transform. The domain is understood to be the domain of the forward transform. The result is a pointer to the created descriptor. This function is slightly different from the "initialization" function [DftiCommitDescriptorDM](#) in a more traditional software packages or libraries used for computing the FFT. This function does not perform any significant computation work, such as twiddle factors computation, because the default configuration settings can still be changed using the function [DftiSetValueDM](#).

The value of the parameter *vl* is specified through named constants `DFTI_SINGLE` and `DFTI_DOUBLE`. It corresponds to precision of input data, output data, and computation. A setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The parameter *dim* is a simple positive integer indicating the dimension of the transform.

For one-dimensional transforms, length is a single integer value of the parameter *size* having type `MKL_LONG`; for multi-dimensional transforms, length is supplied with the parameter *sizes*, which is an array of integers having type `MKL_LONG`.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. In this case, the pointer to the created descriptor handle is stored in *handle*. If the function fails, it returns a value of another error class constant (for the list of constants, refer to [Error Codes](#)).

Prototype

```
MKL_LONG DftiCreateDescriptorDM(MPI_Comm,DFTI_DESCRIPTOR_DM_HANDLE*,
    enum DFTI_CONFIG_VALUE,enum DFTI_CONFIG_VALUE,MKL_LONG,...);
```

DftiCommitDescriptorDM

Performs all initialization for the actual FFT computation.

Syntax

```
status = DftiCommitDescriptorDM(handle);
```

Include Files

- `mkl_cdft.h`

Input Parameters

<i>handle</i>	The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM .
---------------	--

Description

The cluster FFT interface requires a function that completes initialization of a previously created descriptor before the descriptor can be used for FFT computations in a particular MPI process. The [DftiCommitDescriptorDM](#) function performs all initialization that facilitates the actual FFT computation. For the current implementation, it may involve exploring many different factorizations of the input length to search for a highly efficient computation method.

Any changes of configuration parameters of a committed descriptor via the set value function (see [Descriptor Configuration Functions](#)) requires a re-committal of the descriptor before a computation function can be invoked. Typically, this committal function is called right before a computation function call (see [FFT Computation Functions](#)).

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to [Error Codes](#)).

Prototype

```
MKL_LONG DftiCommitDescriptorDM(DFTI_DESCRIPTOR_DM_HANDLE handle);
```

DftiFreeDescriptorDM

Frees memory allocated for a descriptor.

Syntax

```
status = DftiFreeDescriptorDM(&handle);
```

Include Files

- `mkl_cdft.h`

Input Parameters

handle The descriptor handle. Must be valid, that is, created in a call to [DftiCreateDescriptorDM](#).

Output Parameters

handle The descriptor handle. Memory allocated for the handle is released on output.

Description

This function frees up all memory allocated for a descriptor in a particular MPI process. Call the `DftiFreeDescriptorDM` function to delete the descriptor handle. Upon successful completion of `DftiFreeDescriptorDM` the descriptor handle is no longer valid.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to [Error Codes](#)).

Prototype

```
MKL_LONG DftiFreeDescriptorDM(DFTI_DESCRIPTOR_DM_HANDLE *handle);
```

Cluster FFT Computation Functions

There are two functions in this category: compute the forward transform and compute the backward transform.

DftiComputeForwardDM*Computes the forward FFT.*

Syntax

```
status = DftiComputeForwardDM(handle, in_X, out_X);
status = DftiComputeForwardDM(handle, in_out_X);
```

Include Files

- mkl_cdft.h

Input Parameters

<i>handle</i>	The descriptor handle.
<i>in_X, in_out_X</i>	Local part of input data. Array of real or complex values (depending on the forward domain type). Refer to Distributing Data among Processes on how to allocate and initialize the array.

Output Parameters

<i>out_X, in_out_X</i>	Local part of output data. Array of complex values. Refer to Distributing Data among Processes on how to allocate the array.
------------------------	--

Description

The `DftiComputeForwardDM` function computes the forward FFT. Forward FFT is the transform using the factor $e^{-i2\pi/n}$.

Before you call the function, the valid descriptor, created by `DftiCreateDescriptorDM`, must be configured and committed using the `DftiCommitDescriptorDM` function.

The computation is carried out by an internal call to the `DftiComputeForward` function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of `DftiComputeForward`.

The local part of input data, as well as the local part of the output data, is an appropriate sequence of real or complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See [Distributing Data Among Processes](#) for details.

Refer to [Configuration Settings](#) for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter `DFTI_PRECISION` determines the precision of input data, output data, and transform: a setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The configuration parameter `DFTI_PLACEMENT` informs the function whether the computation should be in-place. If the value of this parameter is `DFTI_INPLACE` (default), you must call the function with two parameters, otherwise you must supply three parameters. If `DFTI_PLACEMENT = DFTI_INPLACE` and three parameters are supplied, then the third parameter is ignored.

Caution

Even in case of an out-of-place transform, local array of input data *in_X* may be changed. To save data, make its copy before calling `DftiComputeForwardDM`.

In case of an in-place transform, `DftiComputeForwardDM` dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.

NOTE

You can specify your own workspace of the same size through the configuration parameter `CDFT_WORKSPACE` to avoid redundant memory allocation.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to [Error Codes](#)).

Prototype

```
MKL_LONG DftiComputeForwardDM(DFTI_DESCRIPTOR_DM_HANDLE handle, void *in_X,...);
```

DftiComputeBackwardDM

Computes the backward FFT.

Syntax

```
status = DftiComputeBackwardDM(handle, in_X, out_X);
```

```
status = DftiComputeBackwardDM(handle, in_out_X);
```

Include Files

- `mkl_cdft.h`

Input Parameters

<i>handle</i>	The descriptor handle.
<i>in_X, in_out_X</i>	Local part of input data. Array of complex values. Refer to Distributing Data among Processes on how to allocate and initialize the array.

Output Parameters

<i>out_X, in_out_X</i>	Local part of output data. Array of real or complex values (depending on the forward domain type. Refer to Distributing Data among Processes on how to allocate the array.
------------------------	--

Description

The `DftiComputeBackwardDM` function computes the backward FFT. Backward FFT is the transform using the factor $e^{i2\pi/n}$.

Before you call the function, the valid descriptor, created by `DftiCreateDescriptorDM`, must be configured and committed using the `DftiCommitDescriptorDM` function.

The computation is carried out by an internal call to the `DftiComputeBackward` function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of `DftiComputeBackward`.

The local part of input data, as well as the local part of the output data, is an appropriate sequence of real or complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See [Distributing Data among Processes](#) for details.

Refer to [Configuration Settings](#) for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter `DFTI_PRECISION` determines the precision of input data, output data, and transform: a setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The configuration parameter `DFTI_PLACEMENT` informs the function whether the computation should be in-place. If the value of this parameter is `DFTI_INPLACE` (default), you must call the function with two parameters, otherwise you must supply three parameters. If `DFTI_PLACEMENT = DFTI_INPLACE` and three parameters are supplied, then the third parameter is ignored.

Caution

Even in case of an out-of-place transform, local array of input data `in_X` may be changed. To save data, make its copy before calling `DftiComputeBackwardDM`.

In case of an in-place transform, `DftiComputeBackwardDM` dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.

NOTE

You can specify your own workspace of the same size through the configuration parameter `CDFT_WORKSPACE` to avoid redundant memory allocation.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to [Error Codes](#)).

Prototype

```
MKL_LONG DftiComputeBackwardDM(DFTI_DESCRIPTOR_DM_HANDLE handle, void *in_X,...);
```

Cluster FFT Descriptor Configuration Functions

There are two functions in this category: the value-setting function `DftiSetValueDM` sets one particular configuration parameter to an appropriate value, and the value-getting function `DftiGetValueDM` reads the value of one particular configuration parameter.

Some configuration parameters used by cluster FFT functions originate from the conventional FFT interface (see [Configuration Settings](#) for details).

Other configuration parameters are specific to the cluster FFT. Integer values of these parameters have type `MKL_LONG`. The exact type of the configuration parameters being floating-point scalars is `float` or `double` (matching `DFTI_PRECISION`). The configuration parameters whose values are named constants have the `enum` type. They are defined in the `mk1_cdft.h` header file.

The names of the configuration parameters specific to the cluster FFT interface have the `CDFT` prefix.

DftiSetValueDM

Sets one particular configuration parameter with the specified configuration value.

Syntax

```
status = DftiSetValueDM (handle, param, value);
```

Include Files

- `mkl_cdft.h`

Input Parameters

<i>handle</i>	The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM .
<i>param</i>	Name of a parameter to be set up in the descriptor handle. See Table "Settable Configuration Parameters" for the list of available parameters.
<i>value</i>	Value of the parameter.

Description

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in the table below, and the configuration value must have the corresponding type. See [Configuration Settings](#) for details of the meaning of each setting and for possible values of the parameters whose values are named constants.

Settable Configuration Parameters

Parameter Name	Data Type	Description	Default Value
<code>DFTI_FORWARD_SCALE</code>	Floating-point scalar	Scale factor of forward transform.	1.0
<code>DFTI_BACKWARD_SCALE</code>	Floating-point scalar	Scale factor of backward transform.	1.0
<code>DFTI_PLACEMENT</code>	Named constant	Placement of the computation result.	<code>DFTI_INPLACE</code>
<code>DFTI_ORDERING</code>	Named constant	Scrambling of data order.	<code>DFTI_ORDERED</code>
<code>CDFT_WORKSPACE</code>	Array of an appropriate type	Auxiliary buffer, a user-defined workspace. Enables saving memory during in-place computations.	NULL (allocate workspace dynamically).
<code>DFTI_PACKED_FORMAT</code>	Named constant	Packed format for storing conjugate-even sequence (in the case of a real forward domain).	<ul style="list-style-type: none"> • <code>DFTI_PERM_FORMAT</code> — default and the only available value for one-dimensional transforms • <code>DFTI_CCE_FORMAT</code> — default and the only available value for multi-dimensional transforms

Parameter Name	Data Type	Description	Default Value
DFTI_TRANSPOSE	Named constant	This parameter determines how the output data is located for multi-dimensional transforms. If the parameter value is <code>DFTI_NONE</code> , the data is located in a usual manner described in this document. If the value is <code>DFTI_ALLOW</code> , the last (first) global transposition is not performed for a forward (backward) transform.	<code>DFTI_NONE</code>

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to [Error Codes](#)).

Prototype

```
MKL_LONG DftiSetValueDM(DFTI_DESCRIPTOR_DM_HANDLE handle, int param,...);
```

DftiGetValueDM

Gets the value of one particular configuration parameter.

Syntax

```
status = DftiGetValueDM(handle, param, &value);
```

Include Files

- `mkl_cdft.h`

Input Parameters

<i>handle</i>	The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM .
<i>param</i>	Name of a parameter to be retrieved from the descriptor. See Table "Retrievable Configuration Parameters" for the list of available parameters.

Output Parameters

<i>value</i>	Value of the parameter.
--------------	-------------------------

Description

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in the table below, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type. Possible values of the named constants can be found in [Table "Configuration Parameters"](#) and relevant subsections of the [Configuration Settings](#) section.

Retrievable Configuration Parameters

Parameter Name	Data Type	Description
DFTI_PRECISION	Named constant	Precision of computation, input data and output data.
DFTI_DIMENSION	Integer scalar	Dimension of the transform
DFTI_LENGTHS	Array of integer values	Array of lengths of the transform. Number of lengths corresponds to the dimension of the transform.
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor of forward transform.
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor of backward transform.
DFTI_PLACEMENT	Named constant	Placement of the computation result.
DFTI_COMMIT_STATUS	Named constant	Shows whether descriptor has been committed.
DFTI_FORWARD_DOMAIN	Named constant	Forward domain of transforms, has the value of <code>DFTI_COMPLEX</code> or <code>DFTI_REAL</code> .
DFTI_ORDERING	Named constant	Scrambling of data order.
CDFT_MPI_COMM	Type of MPI communicator	MPI communicator used for transforms.
CDFT_LOCAL_SIZE	Integer scalar	Necessary size of input, output, and buffer arrays in data elements.
CDFT_LOCAL_X_START	Integer scalar	Row/element number of the global array that corresponds to the first row/element of the local array. For more information, see Distributing Data among Processes .
CDFT_LOCAL_NX	Integer scalar	The number of rows/elements of the global array stored in the local array. For more information, see Distributing Data among Processes .
CDFT_LOCAL_OUT_X_START	Integer scalar	Element number of the appropriate global array that corresponds to the first element of the input or output local array in a 1D case. For details, see Distributing Data among Processes .
CDFT_LOCAL_OUT_NX	Integer scalar	The number of elements of the appropriate global array that are stored in the input or output local array in a 1D case. For details, see Distributing Data among Processes .

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to [Error Codes](#)).

Prototype

```
MKL_LONG DftiGetValueDM(DFTI_DESCRIPTOR_DM_HANDLE handle, int param,...);
```

Error Codes

All the cluster FFT functions return an integer value denoting the status of the operation. These values are identified by named constants. Each function returns `DFTI_NO_ERROR` if no errors were encountered during execution. Otherwise, a function generates an error code. In addition to FFT error codes, the cluster FFT interface has its own ones. The named constants specific to the cluster FFT interface have the `CDFT` prefix in their names. [Table "Error Codes that Cluster FFT Functions Return"](#) lists error codes that the cluster FFT functions may return.

Error Codes that Cluster FFT Functions Return

Named Constants	Comments
<code>DFTI_NO_ERROR</code>	No error.
<code>DFTI_MEMORY_ERROR</code>	Usually associated with memory allocation.
<code>DFTI_INVALID_CONFIGURATION</code>	Invalid settings of one or more configuration parameters.
<code>DFTI_INCONSISTENT_CONFIGURATION</code>	Inconsistent configuration or input parameters.
<code>DFTI_NUMBER_OF_THREADS_ERROR</code>	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function).
<code>DFTI_MULTITHREADED_ERROR</code>	Usually associated with a value that OMP routines return in case of errors.
<code>DFTI_BAD_DESCRIPTOR</code>	Descriptor is unusable for computation.
<code>DFTI_UNIMPLEMENTED</code>	Unimplemented legitimate settings; implementation dependent.
<code>DFTI_MKL_INTERNAL_ERROR</code>	Internal library error.
<code>DFTI_1D_LENGTH_EXCEEDS_INT32</code>	Length of one of the dimensions exceeds $2^{32} - 1$ (4 bytes).
<code>CDFT_SPREAD_ERROR</code>	Data cannot be distributed (For more information, see Distributing Data among Processes.)
<code>CDFT_MPI_ERROR</code>	MPI error. Occurs when calling MPI.

PBLAS Routines

Intel® oneAPI Math Kernel Library implements the PBLAS (Parallel Basic Linear Algebra Subprograms) routines from the ScaLAPACK package for distributed-memory architecture. PBLAS is intended for using in vector-vector, matrix-vector, and matrix-matrix operations to simplify the parallelization of linear codes. The design of PBLAS is as consistent as possible with that of the BLAS. The routine descriptions are arranged in several sections according to the PBLAS level of operation:

- [PBLAS Level 1 Routines](#) (distributed vector-vector operations)
- [PBLAS Level 2 Routines](#) (distributed matrix-vector operations)
- [PBLAS Level 3 Routines](#) (distributed matrix-matrix operations)

Each section presents the routine and function group descriptions in alphabetical order by the routine group name; for example, the `p?asum` group, the `p?axpy` group. The question mark in the group name corresponds to a character indicating the data type (`s`, `d`, `c`, and `z` or their combination); see [Routine Naming Conventions](#).

NOTE

PBLAS routines are provided only with Intel® oneAPI Math Kernel Library (oneMKL) versions for Linux* and Windows* OSs.

Generally, PBLAS runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of PBLAS optimized for the target architecture. The Intel® oneAPI Math Kernel Library (oneMKL) version of PBLAS is optimized for Intel® processors. For the detailed system and environment requirements see *Intel® oneAPI Math Kernel Library (oneMKL) Release Notes* and *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

For full reference on PBLAS routines and related information, see http://www.netlib.org/scalapack/html/pblas_qref.html.

Product and Performance Information

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

Notice revision #20201201

PBLAS Routines Overview

The model of the computing environment for PBLAS is represented as a one-dimensional array of processes or also a two-dimensional process grid. To use PBLAS, all global matrices or vectors must be distributed on this array or grid prior to calling the PBLAS routines.

PBLAS uses the two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, as well as gives the opportunity to use PBLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global array and its corresponding process and memory location is contained in the so called *array descriptor* associated with each global array. [Table "Content of the array descriptor for dense matrices"](#) gives an example of an array descriptor structure.

Content of Array Descriptor for Dense Matrices

Array Element #	Name	Definition
1	<i>dtype</i>	Descriptor type (=1 for dense matrices)
2	<i>ctxt</i>	BLACS context handle for the process grid
3	<i>m</i>	Number of rows in the global array
4	<i>n</i>	Number of columns in the global array
5	<i>mb</i>	Row blocking factor
6	<i>nb</i>	Column blocking factor
7	<i>rsrc</i>	Process row over which the first row of the global array is distributed
8	<i>csrc</i>	Process column over which the first column of the global array is distributed
9	<i>lld</i>	Leading dimension of the local array

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by `LOCr()` and `LOCc()`, respectively. To compute these numbers, you can use the ScaLAPACK tool routine `numroc`.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix of the global matrix *A*, which is contained in the global subarray `sub(A)`, defined by the following 6 values (for dense matrices):

<i>m</i>	The number of rows of <code>sub(A)</code>
<i>n</i>	The number of columns of <code>sub(A)</code>
<i>a</i>	A pointer to the local array containing the entire global array <i>A</i>
<i>ia</i>	The row index of <code>sub(A)</code> in the global array
<i>ja</i>	The column index of <code>sub(A)</code> in the global array
<i>desca</i>	The array descriptor for the global array <i>A</i>

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

PBLAS Routine Naming Conventions

The naming convention for PBLAS routines is similar to that used for BLAS routines (see [Routine Naming Conventions](#)). A general rule is that each routine name in PBLAS, which has a BLAS equivalent, is simply the BLAS name prefixed by initial letter *p* that stands for "parallel".

The Intel® oneAPI Math Kernel Library (oneMKL) PBLAS routine names have the following structure:

```
p <character> <name> <mod> ( )
```

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

<i>s</i>	real, single precision
<i>c</i>	complex, single precision
<i>d</i>	real, double precision
<i>z</i>	complex, double precision
<i>i</i>	integer

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

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

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

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

<i>ge</i>	general matrix
<i>sy</i>	symmetric matrix
<i>he</i>	Hermitian matrix
<i>tr</i>	triangular matrix

In PBLAS level 3, the *<name>=tran* indicates the transposition of the matrix.

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

<i>c</i>	conjugated vector
<i>u</i>	unconjugated vector

The PBLAS level 2 names can have the following additional characters in the *<mod>* field:

<i>mv</i>	matrix-vector product
-----------	-----------------------

sv	solving a system of linear equations with matrix-vector operations
r	rank-1 update of a matrix
r2	rank-2 update of a matrix.

The PBLAS level 3 names can have the following additional characters in the `<mod>` field:

mm	matrix-matrix product
sm	solving a system of linear equations with matrix-matrix operations
rk	rank- <i>k</i> update of a matrix
r2k	rank-2 <i>k</i> update of a matrix.

The examples below show how to interpret PBLAS routine names:

pddot	<code><p> <d> <dot></code> : double-precision real distributed vector-vector dot product
pcdotc	<code><p> <c> <dot> <c></code> : complex distributed vector-vector dot product, conjugated
pscasum	<code><p> <sc> <asum></code> : sum of magnitudes of distributed vector elements, single precision real output and single precision complex input
pcdotu	<code><p> <c> <dot> <u></code> : distributed vector-vector dot product, unconjugated, complex
psgemv	<code><p> <s> <ge> <mv></code> : distributed matrix-vector product, general matrix, single precision
pztrmm	<code><p> <z> <tr> <mm></code> : distributed matrix-matrix product, triangular matrix, double-precision complex.

Product and Performance Information

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

Notice revision #20201201

PBLAS Level 1 Routines

PBLAS Level 1 includes routines and functions that perform distributed vector-vector operations. [Table "PBLAS Level 1 Routine Groups and Their Data Types"](#) lists the PBLAS Level 1 routine groups and the data types associated with them.

PBLAS Level 1 Routine Groups and Their Data Types

Routine or Function Group	Data Types	Description
p?amax	s, d, c, z	Calculates an index of the distributed vector element with maximum absolute value
p?asum	s, d, sc, dz	Calculates sum of magnitudes of a distributed vector
p?axpy	s, d, c, z	Calculates distributed vector-scalar product
p?copy	s, d, c, z	Copies a distributed vector
p?dot	s, d	Calculates a dot product of two distributed real vectors

Routine or Function Group	Data Types	Description
<code>p?dotc</code>	c, z	Calculates a dot product of two distributed complex vectors, one of them is conjugated
<code>p?dotu</code>	c, z	Calculates a dot product of two distributed complex vectors
<code>p?nrm2</code>	s, d, sc, dz	Calculates the 2-norm (Euclidean norm) of a distributed vector
<code>p?scal</code>	s, d, c, z, cs, zd	Calculates a product of a distributed vector by a scalar
<code>p?swap</code>	s, d, c, z	Swaps two distributed vectors

p?amax

Computes the global index of the element of a distributed vector with maximum absolute value.

Syntax

```
void psamax (const MKL_INT *n , float *amax , MKL_INT *indx , const float *x , const
MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pdamax (const MKL_INT *n , double *amax , MKL_INT *indx , const double *x , const
MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pcamax (const MKL_INT *n , MKL_Complex8 *amax , MKL_INT *indx , const MKL_Complex8
*x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT
*incx );

void pzamax (const MKL_INT *n , MKL_Complex16 *amax , MKL_INT *indx , const
MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx );
```

Include Files

- `mkl_pblas.h`

Description

The functions `p?amax` compute global index of the maximum element in absolute value of a distributed vector `sub(x)`,

where `sub(x)` denotes $X(ix, jx:jx+n-1)$ if `incx=m_x`, and $X(ix: ix+n-1, jx)$ if `incx= 1`.

Input Parameters

<code>n</code>	(global) The length of distributed vector <code>sub(x)</code> , $n \geq 0$.
<code>x</code>	(local) Array, size $(jx-1) * m_x + ix + (n-1) * \text{abs}(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<code>ix, jx</code>	(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.

<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.

Output Parameters

<i>amax</i>	(global). Maximum absolute value (magnitude) of elements of the distributed vector only in its scope.
<i>indx</i>	(global) The global index of the maximum element in absolute value of the distributed vector $\text{sub}(x)$ only in its scope.

p?asum

Computes the sum of magnitudes of elements of a distributed vector.

Syntax

```
void psasum (const MKL_INT *n , float *asum , const float *x , const MKL_INT *ix ,
const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pdasum (const MKL_INT *n , double *asum , const double *x , const MKL_INT *ix ,
const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pscasum (const MKL_INT *n , float *asum , const MKL_Complex8 *x , const MKL_INT
*ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pdzasum (const MKL_INT *n , double *asum , const MKL_Complex16 *x , const MKL_INT
*ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );
```

Include Files

- mkl_pblas.h

Description

The functions `p?asum` compute the sum of the magnitudes of elements of a distributed vector $\text{sub}(x)$, where $\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$.

Input Parameters

<i>n</i>	(global) The length of distributed vector $\text{sub}(x)$, $n \geq 0$.
<i>x</i>	(local) Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(X)$, respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix X .

incx (global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . *incx* must not be zero.

Output Parameters

asum (local) and *pascalum*.
Contains the sum of magnitudes of elements of the distributed vector only in its scope.

p?axpy

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

Syntax

```
void psaxpy (const MKL_INT *n , const float *a , const float *x , const MKL_INT *ix ,
const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , float *y , const
MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pdaxpy (const MKL_INT *n , const double *a , const double *x , const MKL_INT *ix ,
const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , double *y , const
MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pcaxpy (const MKL_INT *n , const MKL_Complex8 *a , const MKL_Complex8 *x , const
MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx ,
MKL_Complex8 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const
MKL_INT *incy );

void pzaxpy (const MKL_INT *n , const MKL_Complex16 *a , const MKL_Complex16 *x , const
MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx ,
MKL_Complex16 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const
MKL_INT *incy );
```

Include Files

- `mkl_pblas.h`

Description

The `p?axpy` routines perform the following operation with distributed vectors:

```
sub(y) := sub(y) + a*sub(x)
```

where:

a is a scalar;

$\text{sub}(x)$ and $\text{sub}(y)$ are *n*-element distributed vectors.

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$;

$\text{sub}(y)$ denotes $Y(iy, jy:jy+n-1)$ if $incy=m_y$, and $Y(iy: iy+n-1, jy)$ if $incy= 1$.

Input Parameters

n (global) The length of distributed vectors, $n \geq 0$.

a (local)

	Specifies the scalar a .
x	(local) Array, size $(jx-1)*m_x + ix + (n-1)*abs(incx)$. This array contains the entries of the distributed vector $sub(x)$.
ix, jx	(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(X)$, respectively.
$descx$	(global and local) array of dimension 9. The array descriptor of the distributed matrix X .
$incx$	(global) Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.
y	(local) Array, size $(jy-1)*m_y + iy + (n-1)*abs(incy)$. This array contains the entries of the distributed vector $sub(y)$.
iy, jy	(global) The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $sub(Y)$, respectively.
$descy$	(global and local) array of dimension 9. The array descriptor of the distributed matrix Y .
$incy$	(global) Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . $incy$ must not be zero.

Output Parameters

y	Overwritten by $sub(y) := sub(y) + a*sub(x)$.
-----	--

p?copy

Copies one distributed vector to another vector.

Syntax

```
void picopy (const MKL_INT *n , const MKL_INT *x , const MKL_INT *ix , const MKL_INT
*jx , const MKL_INT *descx , const MKL_INT *incx , MKL_INT *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pscopy (const MKL_INT *n , const float *x , const MKL_INT *ix , const MKL_INT
*jx , const MKL_INT *descx , const MKL_INT *incx , float *y , const MKL_INT *iy , const
MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pdcopy (const MKL_INT *n , const double *x , const MKL_INT *ix , const MKL_INT
*jx , const MKL_INT *descx , const MKL_INT *incx , double *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pccopy (const MKL_INT *n , const MKL_Complex8 *x , const MKL_INT *ix , const
MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , MKL_Complex8 *y , const
MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pzcopy (const MKL_INT *n , const MKL_Complex16 *x , const MKL_INT *ix , const
MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , MKL_Complex16 *y , const
MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

Include Files

- `mkl_pblas.h`

Description

The `p?copy` routines perform a copy operation with distributed vectors defined as

```
sub(y) = sub(x),
```

where `sub(x)` and `sub(y)` are n -element distributed vectors.

`sub(x)` denotes $X(ix, jx:jx+n-1)$ if `incx=m_x`, and $X(ix: ix+n-1, jx)$ if `incx= 1`;

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if `incy=m_y`, and $Y(iy: iy+n-1, jy)$ if `incy= 1`.

Input Parameters

n	(global) The length of distributed vectors, $n \geq 0$.
x	(local) Array, size $(jx-1) * m_x + ix + (n-1) * \text{abs}(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
ix, jx	(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.
$descx$	(global and local) array of dimension 9. The array descriptor of the distributed matrix X .
$incx$	(global) Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and <code>m_x</code> . <code>incx</code> must not be zero.
y	(local) Array, size $(jy-1) * m_y + iy + (n-1) * \text{abs}(incy)$. This array contains the entries of the distributed vector <code>sub(y)</code> .
iy, jy	(global) The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix <code>sub(Y)</code> , respectively.
$descy$	(global and local) array of dimension 9. The array descriptor of the distributed matrix Y .
$incy$	(global) Specifies the increment for the elements of <code>sub(y)</code> . Only two values are supported, namely 1 and <code>m_y</code> . <code>incy</code> must not be zero.

Output Parameters

y	Overwritten with the distributed vector <code>sub(x)</code> .
-----	---

p?dot

Computes the dot product of two distributed real vectors.

Syntax

```
void psdot (const MKL_INT *n , float *dot , const float *x , const MKL_INT *ix , const
MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , const float *y , const
MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pddot (const MKL_INT *n , double *dot , const double *x , const MKL_INT *ix ,
const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , const double *y ,
const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

Include Files

- mkl_pblas.h

Description

The `?dot` functions compute the dot product *dot* of two distributed real vectors defined as

$$dot = sub(x)' * sub(y)$$

where `sub(x)` and `sub(y)` are *n*-element distributed vectors.

`sub(x)` denotes $X(ix, jx:jx+n-1)$ if `incx=mx`, and $X(ix: ix+n-1, jx)$ if `incx= 1`;

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if `incy=my`, and $Y(iy: iy+n-1, jy)$ if `incy= 1`.

Input Parameters

<i>n</i>	(global) The length of distributed vectors, $n \geq 0$.
<i>x</i>	(local) Array, size $(jx-1) * m_x + ix + (n-1) * abs(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and <code>m_x</code> . <i>incx</i> must not be zero.
<i>y</i>	(local) Array, size $(jy-1) * m_y + iy + (n-1) * abs(incy)$. This array contains the entries of the distributed vector <code>sub(y)</code> .
<i>iy, jy</i>	(global) The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <code>sub(Y)</code> , respectively.
<i>descy</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) Specifies the increment for the elements of <code>sub(y)</code> . Only two values are supported, namely 1 and <code>m_y</code> . <i>incy</i> must not be zero.

Output Parameters

dot

(local)

Dot product of `sub(x)` and `sub(y)` only in their scope.

p?dotc

Computes the dot product of two distributed complex vectors, one of them is conjugated.

Syntax

```
void pcdotc (const MKL_INT *n , MKL_Complex8 *dotc , const MKL_Complex8 *x , const
MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , const
MKL_Complex8 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const
MKL_INT *incy );
```

```
void pzdadc (const MKL_INT *n , MKL_Complex16 *dotc , const MKL_Complex16 *x , const
MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , const
MKL_Complex16 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const
MKL_INT *incy );
```

Include Files

- `mkl_pblas.h`

Description

The `p?dotc` functions compute the dot product *dotc* of two distributed vectors, with one vector conjugated:

```
dotc = conjg(sub(x)')*sub(y)
```

where `sub(x)` and `sub(y)` are *n*-element distributed vectors.

`sub(x)` denotes `X(ix, jx:jx+n-1)` if `incx=m_x`, and `X(ix: ix+n-1, jx)` if `incx= 1`;

`sub(y)` denotes `Y(iy, jy:jy+n-1)` if `incy=m_y`, and `Y(iy: iy+n-1, jy)` if `incy= 1`.

Input Parameters

n

(global) The length of distributed vectors, $n \geq 0$.

x

(local)

Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$.

This array contains the entries of the distributed vector `sub(x)`.

ix, jx

(global) The row and column indices in the distributed matrix *X* indicating the first row and the first column of the submatrix `sub(X)`, respectively.

descx

(global and local) array of dimension 9. The array descriptor of the distributed matrix *X*.

incx

(global) Specifies the increment for the elements of `sub(x)`. Only two values are supported, namely 1 and `m_x`. `incx` must not be zero.

y

(local)

Array, size $(jy-1)*m_y + iy+(n-1)*abs(incy)$.

This array contains the entries of the distributed vector `sub(y)`.

`iy, jy`

(global) The row and column indices in the distributed matrix `Y` indicating the first row and the first column of the submatrix `sub(Y)`, respectively.

`descy`

(global and local) array of dimension 9. The array descriptor of the distributed matrix `Y`.

`incy`

(global) Specifies the increment for the elements of `sub(y)`. Only two values are supported, namely 1 and `m_y`. `incy` must not be zero.

Output Parameters

`dotc`

(local)

Dot product of `sub(x)` and `sub(y)` only in their scope.

p?dotu

Computes the dot product of two distributed complex vectors.

Syntax

```
void pcdotu (const MKL_INT *n , MKL_Complex8 *dotu , const MKL_Complex8 *x , const
MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , const
MKL_Complex8 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const
MKL_INT *incy );
```

```
void pzdotu (const MKL_INT *n , MKL_Complex16 *dotu , const MKL_Complex16 *x , const
MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , const
MKL_Complex16 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const
MKL_INT *incy );
```

Include Files

- `mkl_pblas.h`

Description

The `p?dotu` functions compute the dot product `dotu` of two distributed vectors defined as

```
dotu = sub(x)'*sub(y)
```

where `sub(x)` and `sub(y)` are n -element distributed vectors.

`sub(x)` denotes $X(ix, jx:jx+n-1)$ if `incx=m_x`, and $X(ix: ix+n-1, jx)$ if `incx= 1`;

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if `incy=m_y`, and $Y(iy: iy+n-1, jy)$ if `incy= 1`.

Input Parameters

`n`

(global) The length of distributed vectors, $n \geq 0$.

`x`

(local)

Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$.

This array contains the entries of the distributed vector `sub(x)`.

<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <i>sub(X)</i> , respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) Specifies the increment for the elements of <i>sub(x)</i> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.
<i>y</i>	(local) Array, size $(jy-1)*m_y + iy + (n-1)*abs(incy)$. This array contains the entries of the distributed vector <i>sub(y)</i> .
<i>iy, jy</i>	(global) The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <i>sub(Y)</i> , respectively.
<i>descy</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

<i>dotu</i>	(local) Dot product of <i>sub(x)</i> and <i>sub(y)</i> only in their scope.
-------------	--

p?nrm2

Computes the Euclidean norm of a distributed vector.

Syntax

```
void psnrm2 (const MKL_INT *n , float *norm2 , const float *x , const MKL_INT *ix ,
const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pdnrm2 (const MKL_INT *n , double *norm2 , const double *x , const MKL_INT *ix ,
const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pscnrm2 (const MKL_INT *n , float *norm2 , const MKL_Complex8 *x , const MKL_INT
*ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pdznrm2 (const MKL_INT *n , double *norm2 , const MKL_Complex16 *x , const MKL_INT
*ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );
```

Include Files

- mkl_pblas.h

Description

The *p?nrm2* functions compute the Euclidean norm of a distributed vector *sub(x)*,

where *sub(x)* is an *n*-element distributed vector.

sub(x) denotes *X(ix, jx:jx+n-1)* if *incx=m_x*, and *X(ix: ix+n-1, jx)* if *incx= 1*.

Input Parameters

<i>n</i>	(global) The length of distributed vector $\text{sub}(x)$, $n \geq 0$.
<i>x</i>	(local) Array, size $(j_x-1)*m_x + ix + (n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix $\text{sub}(X)$, respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.

Output Parameters

<i>norm2</i>	(local) and <i>pscnrm2</i> . Contains the Euclidean norm of a distributed vector only in its scope.
--------------	--

p?scal

Computes a product of a distributed vector by a scalar.

Syntax

```
void psscal (const MKL_INT *n , const float *a , float *x , const MKL_INT *ix , const
MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pdscal (const MKL_INT *n , const double *a , double *x , const MKL_INT *ix , const
MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pcscal (const MKL_INT *n , const MKL_Complex8 *a , MKL_Complex8 *x , const MKL_INT
*ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pzscal (const MKL_INT *n , const MKL_Complex16 *a , MKL_Complex16 *x , const
MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pcsscal (const MKL_INT *n , const float *a , MKL_Complex8 *x , const MKL_INT *ix ,
const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pzdscale (const MKL_INT *n , const double *a , MKL_Complex16 *x , const MKL_INT
*ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );
```

Include Files

- `mkl_pblas.h`

Description

The *p?scal* routines multiplies a *n*-element distributed vector $\text{sub}(x)$ by the scalar *a*:

```
sub(x) = a*sub(x),
```

where $\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$.

Input Parameters

n	(global) The length of distributed vector $\text{sub}(x)$, $n \geq 0$.
a	(global) and <code>pcsscal</code> Specifies the scalar a .
x	(local) Array, size $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
ix, jx	(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(X)$, respectively.
$descx$	(global and local) array of dimension 9. The array descriptor of the distributed matrix X .
$incx$	(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

Output Parameters

x	Overwritten by the updated distributed vector $\text{sub}(x)$
-----	---

p?swap

Swaps two distributed vectors.

Syntax

```
void psswap (const MKL_INT *n , float *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , const MKL_INT *incx , float *y , const MKL_INT *iy , const
MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pdswap (const MKL_INT *n , double *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , const MKL_INT *incx , double *y , const MKL_INT *iy , const
MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pcswap (const MKL_INT *n , MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT
*jx , const MKL_INT *descx , const MKL_INT *incx , MKL_Complex8 *y , const MKL_INT
*iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pzswap (const MKL_INT *n , MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT
*jx , const MKL_INT *descx , const MKL_INT *incx , MKL_Complex16 *y , const MKL_INT
*iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

Include Files

- `mkl_pblas.h`

Description

Given two distributed vectors $\text{sub}(x)$ and $\text{sub}(y)$, the `p?swap` routines return vectors $\text{sub}(y)$ and $\text{sub}(x)$ swapped, each replacing the other.

Here $\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$;

$\text{sub}(y)$ denotes $Y(iy, jy:jy+n-1)$ if $incy=m_y$, and $Y(iy: iy+n-1, jy)$ if $incy= 1$.

Input Parameters

n	(global) The length of distributed vectors, $n \geq 0$.
x	(local) Array, size $(j_x-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector $sub(x)$.
ix, jx	(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(X)$, respectively.
$descx$	(global and local) array of dimension 9. The array descriptor of the distributed matrix X .
$incx$	(global) Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.
y	(local) Array, size $(j_y-1)*m_y + iy+(n-1)*abs(incy)$. This array contains the entries of the distributed vector $sub(y)$.
iy, jy	(global) The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $sub(Y)$, respectively.
$descy$	(global and local) array of dimension 9. The array descriptor of the distributed matrix Y .
$incy$	(global) Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . $incy$ must not be zero.

Output Parameters

x	Overwritten by distributed vector $sub(y)$.
y	Overwritten by distributed vector $sub(x)$.

PBLAS Level 2 Routines

This section describes PBLAS Level 2 routines, which perform distributed matrix-vector operations. [Table "PBLAS Level 2 Routine Groups and Their Data Types"](#) lists the PBLAS Level 2 routine groups and the data types associated with them.

PBLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
p?gemv	s, d, c, z	Matrix-vector product using a distributed general matrix
p?agemv	s, d, c, z	Matrix-vector product using absolute values for a distributed general matrix
p?ger	s, d	Rank-1 update of a distributed general matrix
p?gerc	c, z	Rank-1 update (conjugated) of a distributed general matrix

Routine Groups	Data Types	Description
p?geru	c, z	Rank-1 update (unconjugated) of a distributed general matrix
p?hemv	c, z	Matrix-vector product using a distributed Hermitian matrix
p?ahemv	c, z	Matrix-vector product using absolute values for a distributed Hermitian matrix
p?her	c, z	Rank-1 update of a distributed Hermitian matrix
p?her2	c, z	Rank-2 update of a distributed Hermitian matrix
p?symv	s, d	Matrix-vector product using a distributed symmetric matrix
p?asymv	s, d	Matrix-vector product using absolute values for a distributed symmetric matrix
p?syr	s, d	Rank-1 update of a distributed symmetric matrix
p?syr2	s, d	Rank-2 update of a distributed symmetric matrix
p?trmv	s, d, c, z	Distributed matrix-vector product using a triangular matrix
p?atrmv	s, d, c, z	Distributed matrix-vector product using absolute values for a triangular matrix
p?trsv	s, d, c, z	Solves a system of linear equations whose coefficients are in a distributed triangular matrix

[p?gemv](#)

Computes a distributed matrix-vector product using a general matrix.

Syntax

```
void psgemv (const char *trans , const MKL_INT *m , const MKL_INT *n , const float
*alpha , const float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca , const float *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT
*descx , const MKL_INT *incx , const float *beta , float *y , const MKL_INT *iy , const
MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

```
void pdgemv (const char *trans , const MKL_INT *m , const MKL_INT *n , const double
*alpha , const double *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca , const double *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT
*descx , const MKL_INT *incx , const double *beta , double *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

```
void pcgemv (const char *trans , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , const MKL_INT *incx , const MKL_Complex8 *beta , MKL_Complex8
*y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT
*incy );
```

```
void pzgemv (const char *trans , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx ,
```

```
const MKL_INT *descx , const MKL_INT *incx , const MKL_Complex16 *beta , MKL_Complex16
*y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT
*incy );
```

Include Files

- mkl_pblas.h

Description

The `p?gemv` routines perform a distributed matrix-vector operation defined as

```
sub(y) := alpha*sub(A)*sub(x) + beta*sub(y),
```

or

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

or

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

where

alpha and *beta* are scalars,

sub(A) is a *m*-by-*n* submatrix, *sub(A)* = *A*(*ia:ia+m-1*, *ja:ja+n-1*),

sub(x) and *sub(y)* are subvectors.

When *trans* = 'N' or 'n', *sub(x)* denotes *X*(*ix*, *jx:jx+n-1*) if *incx* = *m_x*, and *X*(*ix: ix+n-1*, *jx*) if *incx* = 1, *sub(y)* denotes *Y*(*iy*, *jy:jy+m-1*) if *incy* = *m_y*, and *Y*(*iy: iy+m-1*, *jy*) if *incy* = 1.

When *trans* = 'T' or 't', or 'C', or 'c', *sub(x)* denotes *X*(*ix*, *jx:jx+m-1*) if *incx* = *m_x*, and *X*(*ix: ix+m-1*, *jx*) if *incx* = 1, *sub(y)* denotes *Y*(*iy*, *jy:jy+n-1*) if *incy* = *m_y*, and *Y*(*iy: iy+m-1*, *jy*) if *incy* = 1.

Input Parameters

<i>trans</i>	(global) Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>sub(y)</i> := <i>alpha</i> * <i>sub(A)</i> '* <i>sub(x)</i> + <i>beta</i> * <i>sub(y)</i> ; if <i>trans</i> = 'T' or 't', then <i>sub(y)</i> := <i>alpha</i> * <i>sub(A)</i> '* <i>sub(x)</i> + <i>beta</i> * <i>sub(y)</i> ; if <i>trans</i> = 'C' or 'c', then <i>sub(y)</i> := <i>alpha</i> * <i>conjg</i> (<i>subA</i> ')* <i>sub(x)</i> + <i>beta</i> * <i>sub(y)</i> .
<i>m</i>	(global) Specifies the number of rows of the distributed matrix <i>sub(A)</i> , <i>m</i> ≥ 0.
<i>n</i>	(global) Specifies the number of columns of the distributed matrix <i>sub(A)</i> , <i>n</i> ≥ 0.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size (<i>lld_a</i> , <i>LOCq(ja+n-1)</i>). Before entry this array must contain the local pieces of the distributed matrix <i>sub(A)</i> .

<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>x</i>	(local) Array, size $(jx-1)*m_x + ix + (n-1)*abs(incx)$ when <i>trans</i> = 'N' or 'n', and $(jx-1)*m_x + ix + (m-1)*abs(incx)$ otherwise. This array contains the entries of the distributed vector <code>sub(x)</code> .
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <code>sub(x)</code> , respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.
<i>beta</i>	(global) Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <code>sub(y)</code> need not be set on input.
<i>y</i>	(local) Array, size $(jy-1)*m_y + iy + (m-1)*abs(incy)$ when <i>trans</i> = 'N' or 'n', and $(jy-1)*m_y + iy + (n-1)*abs(incy)$ otherwise. This array contains the entries of the distributed vector <code>sub(y)</code> .
<i>iy, jy</i>	(global) The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <code>sub(y)</code> , respectively.
<i>descy</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) Specifies the increment for the elements of <code>sub(y)</code> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated distributed vector <code>sub(y)</code> .
----------	---

p?agemv

Computes a distributed matrix-vector product using absolute values for a general matrix.

Syntax

```
void psagemv (const char *trans , const MKL_INT *m , const MKL_INT *n , const float
*alpha , const float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca , const float *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT
*descx , const MKL_INT *incx , const float *beta , float *y , const MKL_INT *iy , const
MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

```

void pdagemv (const char *trans , const MKL_INT *m , const MKL_INT *n , const double
*alpha , const double *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca , const double *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT
*descx , const MKL_INT *incx , const double *beta , double *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pcagemv (const char *trans , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , const MKL_INT *incx , const MKL_Complex8 *beta , MKL_Complex8
*y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT
*incy );

void pzagemv (const char *trans , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , const MKL_INT *incx , const MKL_Complex16 *beta , MKL_Complex16
*y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT
*incy );

```

Include Files

- mkl_pblas.h

Description

The p?agemv routines perform a distributed matrix-vector operation defined as

$$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)') * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y)),$$

or

$$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)') * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y)),$$

or

$$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{conjg}(\text{sub}(A)')) * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y)),$$

where

alpha and *beta* are scalars,

sub(A) is a *m*-by-*n* submatrix, sub(A) = A(*ia:ia+m-1*, *ja:ja+n-1*),

sub(*x*) and sub(*y*) are subvectors.

When *trans* = 'N' or 'n',

sub(*x*) denotes *X*(*ix:ix*, *jx:jx+n-1*) if *incx* = *m_x*, and

X(*ix:ix+n-1*, *jx:jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy:iy*, *jy:jy+m-1*) if *incy* = *m_y*, and

Y(*iy:iy+m-1*, *jy:jy*) if *incy* = 1.

When *trans* = 'T' or 't', or 'C', or 'c',

sub(*x*) denotes *X*(*ix:ix*, *jx:jx+m-1*) if *incx* = *m_x*, and

X(*ix:ix+m-1*, *jx:jx*) if *incx* = 1,

sub(*y*) denotes *Y*(*iy:iy*, *jy:jy+n-1*) if *incy* = *m_y*, and

Y(*iy:iy+m-1*, *jy:jy*) if *incy* = 1.

Input Parameters

<i>trans</i>	<p>(global) Specifies the operation:</p> <p>if <i>trans</i>= 'N' or 'n', then $\text{sub}(y) := \alpha * \text{sub}(A) * \text{sub}(x) + \beta * \text{sub}(y)$</p> <p>if <i>trans</i>= 'T' or 't', then $\text{sub}(y) := \alpha * \text{sub}(A)' * \text{sub}(x) + \beta * \text{sub}(y)$</p> <p>if <i>trans</i>= 'C' or 'c', then $\text{sub}(y) := \alpha * \text{sub}(A)' * \text{sub}(x) + \beta * \text{sub}(y)$.</p>
<i>m</i>	<p>(global) Specifies the number of rows of the distributed matrix $\text{sub}(A)$, $m \geq 0$.</p>
<i>n</i>	<p>(global) Specifies the number of columns of the distributed matrix $\text{sub}(A)$, $n \geq 0$.</p>
<i>alpha</i>	<p>(global)</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>(local)</p> <p>Array, size $(lld_a, LOCq(ja+n-1))$. Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(A)$.</p>
<i>ia, ja</i>	<p>(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.</p>
<i>desca</i>	<p>(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i>.</p>
<i>x</i>	<p>(local)</p> <p>Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$ when <i>trans</i> = 'N' or 'n', and $(jx-1)*m_x + ix+(m-1)*abs(incx)$ otherwise.</p> <p>This array contains the entries of the distributed vector $\text{sub}(x)$.</p>
<i>ix, jx</i>	<p>(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.</p>
<i>descx</i>	<p>(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i>.</p>
<i>incx</i>	<p>(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x. <i>incx</i> must not be zero.</p>
<i>beta</i>	<p>(global)</p> <p>Specifies the scalar <i>beta</i>. When <i>beta</i> is set to zero, then $\text{sub}(y)$ need not be set on input.</p>
<i>y</i>	<p>(local)</p> <p>Array, size $(jy-1)*m_y + iy+(m-1)*abs(incy)$ when <i>trans</i> = 'N' or 'n', and $(jy-1)*m_y + iy+(n-1)*abs(incy)$ otherwise.</p> <p>This array contains the entries of the distributed vector $\text{sub}(y)$.</p>

<i>iy, jy</i>	(global) The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <i>sub(y)</i> , respectively.
<i>descy</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated distributed vector <i>sub(y)</i> .
----------	---

p?ger

Performs a rank-1 update of a distributed general matrix.

Syntax

```
void psger (const MKL_INT *m , const MKL_INT *n , const float *alpha , const float *x ,
const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx ,
const float *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const
MKL_INT *incy , float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca );
```

```
void pdger (const MKL_INT *m , const MKL_INT *n , const double *alpha , const double
*x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT
*incx , const double *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT
*descy , const MKL_INT *incy , double *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca );
```

Include Files

- `mkl_pblas.h`

Description

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

$$\text{sub}(A) := \alpha * \text{sub}(x) * \text{sub}(y)^T + \text{sub}(A),$$

where:

alpha is a scalar,

sub(A) is a *m*-by-*n* distributed general matrix, *sub(A)*=*A*(*ia:ia+m-1*, *ja:ja+n-1*),

sub(x) is an *m*-element distributed vector, *sub(y)* is an *n*-element distributed vector,

sub(x) denotes *X*(*ix*, *jx:jx+m-1*) if *incx* = *m_x*, and *X*(*ix: ix+m-1*, *jx*) if *incx* = 1,

sub(y) denotes *Y*(*iy*, *jy:jy+n-1*) if *incy* = *m_y*, and *Y*(*iy: iy+n-1*, *jy*) if *incy* = 1.

Input Parameters

<i>m</i>	(global) Specifies the number of rows of the distributed matrix <i>sub(A)</i> , <i>m</i> ≥0.
----------	--

<i>n</i>	(global) Specifies the number of columns of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) Array, size at least $(j_x-1)*m_x + ix + (m-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local) Array, size at least $(j_y-1)*m_y + iy + (n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
<i>descy</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.
<i>a</i>	(local) Array, size $(lld_a, LOCq(ja+n-1))$. Before entry this array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	Overwritten by the updated distributed matrix $\text{sub}(A)$.
----------	---

p?gerc

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

Syntax

```
void pcgerc (const MKL_INT *m , const MKL_INT *n , const MKL_Complex8 *alpha , const
MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , const MKL_Complex8 *y , const MKL_INT *iy , const MKL_INT *jy , const
MKL_INT *descy , const MKL_INT *incy , MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca );
```

```
void pzgerc (const MKL_INT *m , const MKL_INT *n , const MKL_Complex16 *alpha , const
MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , const MKL_Complex16 *y , const MKL_INT *iy , const MKL_INT *jy , const
MKL_INT *descy , const MKL_INT *incy , MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca );
```

Include Files

- mkl_pblas.h

Description

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

$$\text{sub}(A) := \alpha * \text{sub}(x) * \text{conjg}(\text{sub}(y)') + \text{sub}(A),$$

where:

alpha is a scalar,

`sub(A)` is a *m*-by-*n* distributed general matrix, `sub(A) = A(ia:ia+m-1, ja:ja+n-1)`,

`sub(x)` is an *m*-element distributed vector, `sub(y)` is an *n*-element distributed vector,

`sub(x)` denotes `X(ix, jx:jx+m-1)` if `incx = m_x`, and `X(ix: ix+m-1, jx)` if `incx = 1`,

`sub(y)` denotes `Y(iy, jy:jy+n-1)` if `incy = m_y`, and `Y(iy: iy+n-1, jy)` if `incy = 1`.

Input Parameters

<i>m</i>	(global) Specifies the number of rows of the distributed matrix <code>sub(A)</code> , $m \geq 0$.
<i>n</i>	(global) Specifies the number of columns of the distributed matrix <code>sub(A)</code> , $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) Array, size at least $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <code>sub(x)</code> , respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.

<i>y</i>	(local) Array, size at least $(j_y-1)*m_y + i_y + (n-1)*abs(incy)$. This array contains the entries of the distributed vector $sub(y)$.
<i>iy, jy</i>	(global) The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix $sub(y)$, respectively.
<i>descy</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.
<i>a</i>	(local) Array, size at least $(lld_a, LOCq(ja+n-1))$. Before entry this array contains the local pieces of the distributed matrix $sub(A)$.
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	Overwritten by the updated distributed matrix $sub(A)$.
----------	--

p?geru

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

Syntax

```
void pcgeru (const MKL_INT *m , const MKL_INT *n , const MKL_Complex8 *alpha , const
MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , const MKL_Complex8 *y , const MKL_INT *iy , const MKL_INT *jy , const
MKL_INT *descy , const MKL_INT *incy , MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca );

void pzgeru (const MKL_INT *m , const MKL_INT *n , const MKL_Complex16 *alpha , const
MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , const MKL_Complex16 *y , const MKL_INT *iy , const MKL_INT *jy , const
MKL_INT *descy , const MKL_INT *incy , MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca );
```

Include Files

- mkl_pblas.h

Description

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

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

where:

alpha is a scalar,

$\text{sub}(A)$ is a m -by- n distributed general matrix, $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$,

$\text{sub}(x)$ is an m -element distributed vector, $\text{sub}(y)$ is an n -element distributed vector,

$\text{sub}(x)$ denotes $X(ix, jx:jx+m-1)$ if $incx = m_x$, and $X(ix:ix+m-1, jx)$ if $incx = 1$,

$\text{sub}(y)$ denotes $Y(iy, jy:jy+n-1)$ if $incy = m_y$, and $Y(iy:iy+n-1, jy)$ if $incy = 1$.

Input Parameters

m	(global) Specifies the number of rows of the distributed matrix $\text{sub}(A)$, $m \geq 0$.
n	(global) Specifies the number of columns of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
α	(global) Specifies the scalar α .
x	(local) Array, size at least $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
ix, jx	(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
$descx$	(global and local) array of dimension 9. The array descriptor of the distributed matrix X .
$incx$	(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.
y	(local) Array, size at least $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
iy, jy	(global) The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
$descy$	(global and local) array of dimension 9. The array descriptor of the distributed matrix Y .
$incy$	(global) Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . $incy$ must not be zero.
a	(local) Array, size at least $(lld_a, \text{LOCq}(ja+n-1))$. Before entry this array contains the local pieces of the distributed matrix $\text{sub}(A)$.
ia, ja	(global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
$desca$	(global and local) array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

a Overwritten by the updated distributed matrix `sub(A)`.

p?hemv

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

Syntax

```
void pchemv (const char *uplo , const MKL_INT *n , const MKL_Complex8 *alpha , const
MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const
MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , const MKL_Complex8 *beta , MKL_Complex8 *y , const MKL_INT *iy , const
MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pzhemv (const char *uplo , const MKL_INT *n , const MKL_Complex16 *alpha , const
MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const
MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , const MKL_Complex16 *beta , MKL_Complex16 *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

Include Files

- `mkc_pblas.h`

Description

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

$$\text{sub}(y) := \alpha \text{sub}(A) \text{sub}(x) + \beta \text{sub}(y),$$

where:

alpha and *beta* are scalars,

`sub(A)` is a *n*-by-*n* Hermitian distributed matrix, `sub(A)=A(ia:ia+n-1, ja:ja+n-1)`,

`sub(x)` and `sub(y)` are distributed vectors.

`sub(x)` denotes `X(ix, jx:jx+n-1)` if `incx = m_x`, and `X(ix: ix+n-1, jx)` if `incx = 1`,

`sub(y)` denotes `Y(iy, jy:jy+n-1)` if `incy = m_y`, and `Y(iy: iy+n-1, jy)` if `incy = 1`.

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <code>sub(A)</code> is used: If <code>uplo = 'U'</code> or <code>'u'</code> , then the upper triangular part of the <code>sub(A)</code> is used. If <code>uplo = 'L'</code> or <code>'l'</code> , then the low triangular part of the <code>sub(A)</code> is used.
<i>n</i>	(global) Specifies the order of the distributed matrix <code>sub(A)</code> , $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local)

Array, size $(lld_a, LOCq(ja+n-1))$. This array contains the local pieces of the distributed matrix $sub(A)$.

Before entry when $uplo = 'U'$ or $'u'$, the n -by- n upper triangular part of the distributed matrix $sub(A)$ must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of $sub(A)$ is not referenced, and when $uplo = 'L'$ or $'l'$, the n -by- n lower triangular part of the distributed matrix $sub(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $sub(A)$ is not referenced.

ia, ja (global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.

desca (global and local) array of dimension 9. The array descriptor of the distributed matrix A .

x (local)
Array, size at least $(jx-1)*m_x + ix+(n-1)*abs(incx)$.
This array contains the entries of the distributed vector $sub(x)$.

ix, jx (global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(x)$, respectively.

descx (global and local) array of dimension 9. The array descriptor of the distributed matrix X .

incx (global) Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . *incx* must not be zero.

beta (global)
Specifies the scalar *beta*. When *beta* is set to zero, then $sub(y)$ need not be set on input.

y (local)
Array, size at least $(jy-1)*m_y + iy+(n-1)*abs(incy)$.
This array contains the entries of the distributed vector $sub(y)$.

iy, jy (global) The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $sub(y)$, respectively.

descy (global and local) array of dimension 9. The array descriptor of the distributed matrix Y .

incy (global) Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . *incy* must not be zero.

Output Parameters

y Overwritten by the updated distributed vector $sub(y)$.

p?ahemv

Computes a distributed matrix-vector product using absolute values for a Hermitian matrix.

Syntax

```
void pcahemv (const char *uplo , const MKL_INT *n , const MKL_Complex8 *alpha , const
MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const
MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , const MKL_Complex8 *beta , MKL_Complex8 *y , const MKL_INT *iy , const
MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pzahemv (const char *uplo , const MKL_INT *n , const MKL_Complex16 *alpha , const
MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const
MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , const MKL_Complex16 *beta , MKL_Complex16 *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

Include Files

- mkl_pblas.h

Description

The p?ahemv routines perform a distributed matrix-vector operation defined as

$$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)) * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y)),$$

where:

alpha and *beta* are scalars,

sub(A) is a *n*-by-*n* Hermitian distributed matrix, *sub(A)* = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) ,

sub(x) and *sub(y)* are distributed vectors.

sub(x) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,

sub(y) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(A)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(A)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(A)</i> is used.
<i>n</i>	(global) Specifies the order of the distributed matrix <i>sub(A)</i> , <i>n</i> ≥ 0.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size (<i>lld_a</i> , LOCq(<i>ja</i> + <i>n</i> -1)). This array contains the local pieces of the distributed matrix <i>sub(A)</i> . Before entry when <i>uplo</i> = 'U' or 'u', the <i>n</i> -by- <i>n</i> upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of <i>sub(A)</i> is not referenced, and when <i>uplo</i> = 'L' or 'l', the <i>n</i> -by- <i>n</i> lower

triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>x</i>	(local) Array, size at least $(j_x-1)*m_x + ix+(n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>beta</i>	(global) Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then $\text{sub}(y)$ need not be set on input.
<i>y</i>	(local) Array, size at least $(j_y-1)*m_y + iy+(n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
<i>descy</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated distributed vector $\text{sub}(y)$.
----------	---

p?her

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

Syntax

```
void pcher (const char *uplo , const MKL_INT *n , const float *alpha , const
MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca );
```

```
void pzher (const char *uplo , const MKL_INT *n , const double *alpha , const
MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const
MKL_INT *desca );
```

Include Files

- mkl_pblas.h

Description

The `pzher` routines perform a distributed matrix-vector operation defined as

$$\text{sub}(A) := \alpha * \text{sub}(x) * \text{conjg}(\text{sub}(x)') + \text{sub}(A),$$

where:

alpha is a real scalar,

`sub(A)` is a *n*-by-*n* distributed Hermitian matrix, `sub(A)=A(ia:ia+n-1, ja:ja+n-1)`,

`sub(x)` is distributed vector.

`sub(x)` denotes `X(ix, jx:jx+n-1)` if `incx = m_x`, and `X(ix: ix+n-1, jx)` if `incx = 1`.

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <code>sub(A)</code> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <code>sub(A)</code> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <code>sub(A)</code> is used.
<i>n</i>	(global) Specifies the order of the distributed matrix <code>sub(A)</code> , $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) Array, size at least $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <code>X</code> indicating the first row and the first column of the submatrix <code>sub(x)</code> , respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <code>X</code> .
<i>incx</i>	(global) Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>a</i>	(local) Array, size $(lld_a, LOCq(ja+n-1))$. This array contains the local pieces of the distributed matrix <code>sub(A)</code> . Before entry with <i>uplo</i> = 'U' or 'u', the <i>n</i> -by- <i>n</i> upper triangular part of the distributed matrix <code>sub(A)</code> must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of <code>sub(A)</code>

is not referenced, and with `uplo = 'L' or 'l'`, the n -by- n lower triangular part of the distributed matrix `sub(A)` must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of `sub(A)` is not referenced.

`ia, ja`

(global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix `sub(A)`, respectively.

`desca`

(global and local) array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

`a`

With `uplo = 'U' or 'u'`, the upper triangular part of the array a is overwritten by the upper triangular part of the updated distributed matrix `sub(A)`.

With `uplo = 'L' or 'l'`, the lower triangular part of the array a is overwritten by the lower triangular part of the updated distributed matrix `sub(A)`.

p?her2

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

Syntax

```
void pcher2 (const char *uplo , const MKL_INT *n , const MKL_Complex8 *alpha , const
MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , const MKL_Complex8 *y , const MKL_INT *iy , const MKL_INT *jy , const
MKL_INT *descy , const MKL_INT *incy , MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca );
```

```
void pzher2 (const char *uplo , const MKL_INT *n , const MKL_Complex16 *alpha , const
MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx , const MKL_Complex16 *y , const MKL_INT *iy , const MKL_INT *jy , const
MKL_INT *descy , const MKL_INT *incy , MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca );
```

Include Files

- `mkl_pblas.h`

Description

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

$$\text{sub}(A) := \alpha * \text{sub}(x) * \text{conj}(\text{sub}(y)') + \text{conj}(\alpha) * \text{sub}(y) * \text{conj}(\text{sub}(x)') + \text{sub}(A),$$

where:

α is a scalar,

`sub(A)` is a n -by- n distributed Hermitian matrix, `sub(A)=A(ia:ia+n-1, ja:ja+n-1)`,

`sub(x)` and `sub(y)` are distributed vectors.

`sub(x)` denotes $X(ix, jx:jx+n-1)$ if `incx = m_x`, and $X(ix: ix+n-1, jx)$ if `incx = 1`,

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if `incy = m_y`, and $Y(iy: iy+n-1, jy)$ if `incy = 1`.

Input Parameters

<i>uplo</i>	<p>(global) Specifies whether the upper or lower triangular part of the distributed Hermitian matrix $\text{sub}(A)$ is used:</p> <p>If $\text{uplo} = 'U'$ or $'u'$, then the upper triangular part of the $\text{sub}(A)$ is used.</p> <p>If $\text{uplo} = 'L'$ or $'l'$, then the low triangular part of the $\text{sub}(A)$ is used.</p>
<i>n</i>	(global) Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	<p>(global)</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>x</i>	<p>(local)</p> <p>Array, size at least $(j_x-1)*m_x + ix + (n-1)*\text{abs}(incx)$.</p> <p>This array contains the entries of the distributed vector $\text{sub}(x)$.</p>
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	<p>(local)</p> <p>Array, size at least $(j_y-1)*m_y + iy + (n-1)*\text{abs}(incy)$.</p> <p>This array contains the entries of the distributed vector $\text{sub}(y)$.</p>
<i>iy, jy</i>	(global) The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
<i>descy</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix Y .
<i>incy</i>	(global) Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.
<i>a</i>	<p>(local)</p> <p>Array, size $(lld_a, \text{LOCq}(ja+n-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$.</p> <p>Before entry with $\text{uplo} = 'U'$ or $'u'$, the n-by-n upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and with $\text{uplo} = 'L'$ or $'l'$, the n-by-n lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.</p>
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca (global and local) array of dimension 9. The array descriptor of the distributed matrix *A*.

Output Parameters

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

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

p?symv

Computes a distributed matrix-vector product using a symmetric matrix.

Syntax

```
void pssymv (const char *uplo , const MKL_INT *n , const float *alpha , const float
*a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const float *x ,
const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx ,
const float *beta , float *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT
*descy , const MKL_INT *incy );
```

```
void pdsymv (const char *uplo , const MKL_INT *n , const double *alpha , const double
*a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const double *x ,
const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx ,
const double *beta , double *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT
*descy , const MKL_INT *incy );
```

Include Files

- `mkl_pblas.h`

Description

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

$$\text{sub}(y) := \alpha * \text{sub}(A) * \text{sub}(x) + \beta * \text{sub}(y),$$

where:

alpha and *beta* are scalars,

sub(A) is a *n*-by-*n* symmetric distributed matrix, *sub(A)* = *A*(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) ,

sub(x) and *sub(y)* are distributed vectors.

sub(x) denotes *X*(*ix*, *jx*:*jx*+*n*-1) if *incx* = *m_x*, and *X*(*ix*: *ix*+*n*-1, *jx*) if *incx* = 1,

sub(y) denotes *Y*(*iy*, *jy*:*jy*+*n*-1) if *incy* = *m_y*, and *Y*(*iy*: *iy*+*n*-1, *jy*) if *incy* = 1.

Input Parameters

uplo (global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix *sub(A)* is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the *sub(A)* is used.

	If <code>uplo = 'L' or 'l'</code> , then the low triangular part of the <code>sub(A)</code> is used.
<code>n</code>	(global) Specifies the order of the distributed matrix <code>sub(A)</code> , $n \geq 0$.
<code>alpha</code>	(global) Specifies the scalar <code>alpha</code> .
<code>a</code>	(local) Array, size <code>(lld_a, LOCq(ja+n-1))</code> . This array contains the local pieces of the distributed matrix <code>sub(A)</code> . Before entry when <code>uplo = 'U' or 'u'</code> , the n -by- n upper triangular part of the distributed matrix <code>sub(A)</code> must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of <code>sub(A)</code> is not referenced, and when <code>uplo = 'L' or 'l'</code> , the n -by- n lower triangular part of the distributed matrix <code>sub(A)</code> must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of <code>sub(A)</code> is not referenced.
<code>ia, ja</code>	(global) The row and column indices in the distributed matrix <code>A</code> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<code>desca</code>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <code>A</code> .
<code>x</code>	(local) Array, size at least <code>(jx-1)*m_x + ix+(n-1)*abs(incx)</code> . This array contains the entries of the distributed vector <code>sub(x)</code> .
<code>ix, jx</code>	(global) The row and column indices in the distributed matrix <code>X</code> indicating the first row and the first column of the submatrix <code>sub(x)</code> , respectively.
<code>descx</code>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <code>X</code> .
<code>incx</code>	(global) Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and <code>m_x</code> . <code>incx</code> must not be zero.
<code>beta</code>	(global) Specifies the scalar <code>beta</code> . When <code>beta</code> is set to zero, then <code>sub(y)</code> need not be set on input.
<code>y</code>	(local) Array, size at least <code>(jy-1)*m_y + iy+(n-1)*abs(incy)</code> . This array contains the entries of the distributed vector <code>sub(y)</code> .
<code>iy, jy</code>	(global) The row and column indices in the distributed matrix <code>Y</code> indicating the first row and the first column of the submatrix <code>sub(y)</code> , respectively.
<code>descy</code>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <code>Y</code> .
<code>incy</code>	(global) Specifies the increment for the elements of <code>sub(y)</code> . Only two values are supported, namely 1 and <code>m_y</code> . <code>incy</code> must not be zero.

Output Parameters

y Overwritten by the updated distributed vector `sub(y)`.

p?asymv

Computes a distributed matrix-vector product using absolute values for a symmetric matrix.

Syntax

```
void psasymv (const char *uplo , const MKL_INT *n , const float *alpha , const float
*a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const float *x ,
const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx ,
const float *beta , float *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT
*descy , const MKL_INT *incy );

void pdasymv (const char *uplo , const MKL_INT *n , const double *alpha , const double
*a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const double *x ,
const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx ,
const double *beta , double *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT
*descy , const MKL_INT *incy );
```

Include Files

- `mkc_pblas.h`

Description

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

$$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)) * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y)),$$

where:

alpha and *beta* are scalars,

`sub(A)` is a *n*-by-*n* symmetric distributed matrix, `sub(A)=A(ia:ia+n-1, ja:ja+n-1)`,

`sub(x)` and `sub(y)` are distributed vectors.

`sub(x)` denotes `X(ix, jx:jx+n-1)` if `incx = m_x`, and `X(ix: ix+n-1, jx)` if `incx = 1`,

`sub(y)` denotes `Y(iy, jy:jy+n-1)` if `incy = m_y`, and `Y(iy: iy+n-1, jy)` if `incy = 1`.

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix <code>sub(A)</code> is used: If <code>uplo = 'U'</code> or <code>'u'</code> , then the upper triangular part of the <code>sub(A)</code> is used. If <code>uplo = 'L'</code> or <code>'l'</code> , then the low triangular part of the <code>sub(A)</code> is used.
<i>n</i>	(global) Specifies the order of the distributed matrix <code>sub(A)</code> , $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local)

Array, size $(lld_a, LOCq(ja+n-1))$. This array contains the local pieces of the distributed matrix $sub(A)$.

Before entry when $uplo = 'U'$ or $'u'$, the n -by- n upper triangular part of the distributed matrix $sub(A)$ must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of $sub(A)$ is not referenced, and when $uplo = 'L'$ or $'l'$, the n -by- n lower triangular part of the distributed matrix $sub(A)$ must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of $sub(A)$ is not referenced.

ia, ja

(global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $sub(A)$, respectively.

desca

(global and local) array of dimension 9. The array descriptor of the distributed matrix A .

x

(local)

Array, size at least $(jx-1)*m_x + ix+(n-1)*abs(incx)$.

This array contains the entries of the distributed vector $sub(x)$.

ix, jx

(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(x)$, respectively.

descx

(global and local) array of dimension 9. The array descriptor of the distributed matrix X .

incx

(global) Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

beta

(global)

Specifies the scalar $beta$. When $beta$ is set to zero, then $sub(y)$ need not be set on input.

y

(local)

Array, size at least $(jy-1)*m_y + iy+(n-1)*abs(incy)$.

This array contains the entries of the distributed vector $sub(y)$.

iy, jy

(global) The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $sub(y)$, respectively.

descy

(global and local) array of dimension 9. The array descriptor of the distributed matrix Y .

incy

(global) Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . $incy$ must not be zero.

Output Parameters

y

Overwritten by the updated distributed vector $sub(y)$.

p?syrr

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

Syntax

```
void pssyr (const char *uplo , const MKL_INT *n , const float *alpha , const float *x ,
const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx ,
float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca );

void pdsyr (const char *uplo , const MKL_INT *n , const double *alpha , const double
*x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT
*incx , double *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca );
```

Include Files

- mkl_pblas.h

Description

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

$$\text{sub}(A) := \alpha * \text{sub}(x) * \text{sub}(x)' + \text{sub}(A),$$

where:

alpha is a scalar,

sub(A) is a *n*-by-*n* distributed symmetric matrix, *sub(A)*=*A*(*ia:ia+n-1*, *ja:ja+n-1*) ,

sub(x) is distributed vector.

sub(x) denotes *X*(*ix*, *jx:jx+n-1*) if *incx* = *m_x*, and *X*(*ix: ix+n-1*, *jx*) if *incx* = 1,

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(A)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(A)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(A)</i> is used.
<i>n</i>	(global) Specifies the order of the distributed matrix <i>sub(A)</i> , $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) Array, size at least $(jx-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector <i>sub(x)</i> .
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <i>sub(x)</i> , respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) Specifies the increment for the elements of <i>sub(x)</i> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.
<i>a</i>	(local) Array, size $(lld_a, LOCq(ja+n-1))$. This array contains the local pieces of the distributed matrix <i>sub(A)</i> .

Before entry with `uplo = 'U' or 'u'`, the n -by- n upper triangular part of the distributed matrix `sub(A)` must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of `sub(A)` is not referenced, and with `uplo = 'L' or 'l'`, the n -by- n lower triangular part of the distributed matrix `sub(A)` must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of `sub(A)` is not referenced.

`ia, ja`

(global) The row and column indices in the distributed matrix `A` indicating the first row and the first column of the submatrix `sub(A)`, respectively.

`desca`

(global and local) array of dimension 9. The array descriptor of the distributed matrix `A`.

Output Parameters

`a`

With `uplo = 'U' or 'u'`, the upper triangular part of the array `a` is overwritten by the upper triangular part of the updated distributed matrix `sub(A)`.

With `uplo = 'L' or 'l'`, the lower triangular part of the array `a` is overwritten by the lower triangular part of the updated distributed matrix `sub(A)`.

p?syr2

Performs a rank-2 update of a distributed symmetric matrix.

Syntax

```
void pssyr2 (const char *uplo , const MKL_INT *n , const float *alpha , const float
*x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT
*incx , const float *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy ,
const MKL_INT *incy , float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca );
```

```
void pdsyr2 (const char *uplo , const MKL_INT *n , const double *alpha , const double
*x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT
*incx , const double *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT
*descy , const MKL_INT *incy , double *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca );
```

Include Files

- `mkl_pblas.h`

Description

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

$$\text{sub}(A) := \alpha * \text{sub}(x) * \text{sub}(y)' + \alpha * \text{sub}(y) * \text{sub}(x)' + \text{sub}(A),$$

where:

α is a scalar,

`sub(A)` is a n -by- n distributed symmetric matrix, `sub(A)=A(ia:ia+n-1, ja:ja+n-1)`,

`sub(x)` and `sub(y)` are distributed vectors.

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx = m_x$, and $X(ix: ix+n-1, jx)$ if $incx = 1$,

$\text{sub}(y)$ denotes $Y(iy, jy:jy+n-1)$ if $incy = m_y$, and $Y(iy: iy+n-1, jy)$ if $incy = 1$.

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the distributed symmetric matrix $\text{sub}(A)$ is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the $\text{sub}(A)$ is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the $\text{sub}(A)$ is used.
<i>n</i>	(global) Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) Array, size at least $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local) Array, size at least $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
<i>descy</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix Y .
<i>incy</i>	(global) Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.
<i>a</i>	(local) Array, size $(lld_a, LOCq(ja+n-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$. Before entry with <i>uplo</i> = 'U' or 'u', the n -by- n upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the distributed symmetric matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and with <i>uplo</i> = 'L' or 'l', the n -by- n lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the distributed symmetric matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated distributed matrix <i>sub(A)</i> . With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated distributed matrix <i>sub(A)</i> .
----------	--

p?trmv

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

Syntax

```
void pstrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca ,
float *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT
*incx );

void pdtrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const double *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca ,
double *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx );

void pctrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca , MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT
*descx , const MKL_INT *incx );

void pztrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca , MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT
*descx , const MKL_INT *incx );
```

Include Files

- mkl_pblas.h

Description

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

$\text{sub}(x) := \text{sub}(A) * \text{sub}(x)$, or $\text{sub}(x) := \text{sub}(A)^T * \text{sub}(x)$, or $\text{sub}(x) := \text{conjg}(\text{sub}(A)^T) * \text{sub}(x)$,

where:

sub(A) is a *n*-by-*n* unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$,

sub(x) is an *n*-element distributed vector.

sub(x) denotes $X(ix, jx:jx+n-1)$ if *incx* = *m_x*, and $X(ix:ix+n-1, jx)$ if *incx* = 1,

Input Parameters

<i>uplo</i>	<p>(global) Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular:</p> <p>if $\text{uplo} = 'U'$ or $'u'$, then the matrix is upper triangular;</p> <p>if $\text{uplo} = 'L'$ or $'l'$, then the matrix is low triangular.</p>
<i>trans</i>	<p>(global) Specifies the form of $\text{op}(\text{sub}(A))$ used in the matrix equation:</p> <p>if $\text{transa} = 'N'$ or $'n'$, then $\text{sub}(x) := \text{sub}(A) * \text{sub}(x)$;</p> <p>if $\text{transa} = 'T'$ or $'t'$, then $\text{sub}(x) := \text{sub}(A)^T * \text{sub}(x)$;</p> <p>if $\text{transa} = 'C'$ or $'c'$, then $\text{sub}(x) := \text{conjg}(\text{sub}(A)^T) * \text{sub}(x)$.</p>
<i>diag</i>	<p>(global) Specifies whether the matrix $\text{sub}(A)$ is unit triangular:</p> <p>if $\text{diag} = 'U'$ or $'u'$ then the matrix is unit triangular;</p> <p>if $\text{diag} = 'N'$ or $'n'$, then the matrix is not unit triangular.</p>
<i>n</i>	(global) Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>a</i>	<p>(local)</p> <p>Array, size at least $(\text{lld}_a, \text{LOCq}(1, ja+n-1))$.</p> <p>Before entry with $\text{uplo} = 'U'$ or $'u'$, this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.</p> <p>Before entry with $\text{uplo} = 'L'$ or $'l'$, this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.</p> <p>When $\text{diag} = 'U'$ or $'u'$, the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.</p>
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix A .
<i>x</i>	<p>(local)</p> <p>Array, size at least $(jx-1)*m_x + ix + (n-1)*\text{abs}(\text{incx})$.</p> <p>This array contains the entries of the distributed vector $\text{sub}(x)$.</p>
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . incx must not be zero.

Output Parameters

x

Overwritten by the transformed distributed vector `sub(x)`.

p?atrmv

Computes a distributed matrix-vector product using absolute values for a triangular matrix.

Syntax

```
void psatrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const float *alpha , const float *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const float *x , const MKL_INT *ix , const MKL_INT *jx , const
MKL_INT *descx , const MKL_INT *incx , const float *beta , float *y , const MKL_INT
*iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

```
void pdatrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const double *alpha , const double *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const double *x , const MKL_INT *ix , const MKL_INT *jx , const
MKL_INT *descx , const MKL_INT *incx , const double *beta , double *y , const MKL_INT
*iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

```
void pcatrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *x , const MKL_INT *ix , const
MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , const MKL_Complex8 *beta ,
MKL_Complex8 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const
MKL_INT *incy );
```

```
void pzatrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *x , const MKL_INT *ix , const
MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx , const MKL_Complex16 *beta ,
MKL_Complex16 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const
MKL_INT *incy );
```

Include Files

- `mkл_pblas.h`

Description

The `p?atrmv` routines perform one of the following distributed matrix-vector operations defined as

$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)) * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y))$, or

$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{sub}(A)^T) * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y))$, or

$\text{sub}(y) := \text{abs}(\alpha) * \text{abs}(\text{conjg}(\text{sub}(A)^T)) * \text{abs}(\text{sub}(x)) + \text{abs}(\beta * \text{sub}(y))$,

where:

α and β are scalars,

$\text{sub}(A)$ is a n -by- n unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$,

$\text{sub}(x)$ is an n -element distributed vector.

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx = m_x$, and $X(ix: ix+n-1, jx)$ if $incx = 1$.

Input Parameters

<i>uplo</i>	<p>(global) Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular:</p> <p>if $\text{uplo} = 'U'$ or $'u'$, then the matrix is upper triangular;</p> <p>if $\text{uplo} = 'L'$ or $'l'$, then the matrix is low triangular.</p>
<i>trans</i>	<p>(global) Specifies the form of $\text{op}(\text{sub}(A))$ used in the matrix equation:</p> <p>if $\text{trans} = 'N'$ or $'n'$, then $\text{sub}(y) := \alpha * \text{sub}(A) * \text{sub}(x) + \beta * \text{sub}(y)$;</p> <p>if $\text{trans} = 'T'$ or $'t'$, then $\text{sub}(y) := \alpha * \text{sub}(A)' * \text{sub}(x) + \beta * \text{sub}(y)$;</p> <p>if $\text{trans} = 'C'$ or $'c'$, then $\text{sub}(y) := \alpha * \text{conjg}(\text{sub}(A)') * \text{sub}(x) + \beta * \text{sub}(y)$.</p>
<i>diag</i>	<p>(global) Specifies whether the matrix $\text{sub}(A)$ is unit triangular:</p> <p>if $\text{diag} = 'U'$ or $'u'$ then the matrix is unit triangular;</p> <p>if $\text{diag} = 'N'$ or $'n'$, then the matrix is not unit triangular.</p>
<i>n</i>	(global) Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	<p>(global)</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>(local)</p> <p>Array, size at least $(\text{lld_a}, \text{LOCq}(1, \text{ja} + n - 1))$.</p> <p>Before entry with $\text{uplo} = 'U'$ or $'u'$, this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.</p> <p>Before entry with $\text{uplo} = 'L'$ or $'l'$, this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.</p> <p>When $\text{diag} = 'U'$ or $'u'$, the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.</p>
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>x</i>	<p>(local)</p> <p>Array, size at least $(\text{jx} - 1) * m_x + \text{ix} + (n - 1) * \text{abs}(\text{incx})$.</p> <p>This array contains the entries of the distributed vector $\text{sub}(x)$.</p>

<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <i>sub(x)</i> , respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) Specifies the increment for the elements of <i>sub(x)</i> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.
<i>beta</i>	(global) Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <i>sub(y)</i> need not be set on input.
<i>y</i>	(local) Array, size $(jy-1)*m_y + iy + (m-1)*abs(incy)$ when <i>trans</i> = 'N' or 'n', and $(jy-1)*m_y + iy + (n-1)*abs(incy)$ otherwise. This array contains the entries of the distributed vector <i>sub(y)</i> .
<i>iy, jy</i>	(global) The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <i>sub(y)</i> , respectively.
<i>descy</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the transformed distributed vector <i>sub(y)</i> .
----------	---

p?trsv

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

Syntax

```
void pstrsv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca ,
float *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT
*incx );

void pdtrsv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const double *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca ,
double *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const
MKL_INT *incx );

void pctrsv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca , MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT
*descx , const MKL_INT *incx );

void pztrsv (const char *uplo , const char *trans , const char *diag , const MKL_INT
*n , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca , MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT
*descx , const MKL_INT *incx );
```


Include Files

- `mkl_pblas.h`

Description

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

$\text{sub}(A) * \text{sub}(x) = b$, or $\text{sub}(A)' * \text{sub}(x) = b$, or $\text{conjg}(\text{sub}(A)') * \text{sub}(x) = b$,

where:

$\text{sub}(A)$ is a n -by- n unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$,

b and $\text{sub}(x)$ are n -element distributed vectors,

$\text{sub}(x)$ denotes $X(\text{ix}, \text{jx}:\text{jx}+n-1)$ if $\text{incx} = m_x$, and $X(\text{ix}:\text{ix}+n-1, \text{jx})$ if $\text{incx} = 1$.

The routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

Input Parameters

<i>uplo</i>	(global) Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	(global) Specifies the form of the system of equations: if <i>transa</i> = 'N' or 'n', then $\text{sub}(A) * \text{sub}(x) = b$; if <i>transa</i> = 'T' or 't', then $\text{sub}(A)' * \text{sub}(x) = b$; if <i>transa</i> = 'C' or 'c', then $\text{conjg}(\text{sub}(A)') * \text{sub}(x) = b$.
<i>diag</i>	(global) Specifies whether the matrix $\text{sub}(A)$ is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	(global) Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>a</i>	(local) Array, size at least $(\text{lld_a}, \text{LOCq}(1, \text{ja}+n-1))$. Before entry with <i>uplo</i> = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced . When <i>diag</i> = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.

<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>x</i>	(local) Array, size at least $(j_x-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> . Before entry, <code>sub(x)</code> must contain the <i>n</i> -element right-hand side distributed vector <i>b</i> .
<i>ix, jx</i>	(global) The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <code>sub(x)</code> , respectively.
<i>descx</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten with the solution vector.
----------	---------------------------------------

PBLAS Level 3 Routines

The PBLAS Level 3 routines perform distributed matrix-matrix operations. [Table "PBLAS Level 3 Routine Groups and Their Data Types"](#) lists the PBLAS Level 3 routine groups and the data types associated with them.

PBLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
p?geadd	s, d, c, z	Distributed matrix-matrix sum of general matrices
p?tradd	s, d, c, z	Distributed matrix-matrix sum of triangular matrices
p?gemm	s, d, c, z	Distributed matrix-matrix product of general matrices
p?hemm	c, z	Distributed matrix-matrix product, one matrix is Hermitian
p?herk	c, z	Rank-k update of a distributed Hermitian matrix
p?her2k	c, z	Rank-2k update of a distributed Hermitian matrix
p?symm	s, d, c, z	Matrix-matrix product of distributed symmetric matrices
p?syrk	s, d, c, z	Rank-k update of a distributed symmetric matrix
p?syr2k	s, d, c, z	Rank-2k update of a distributed symmetric matrix
p?tran	s, d	Transposition of a real distributed matrix
p?tranc	c, z	Transposition of a complex distributed matrix (conjugated)
p?tranu	c, z	Transposition of a complex distributed matrix

Routine Group	Data Types	Description
p?trmm	s, d, c, z	Distributed matrix-matrix product, one matrix is triangular
p?trsm	s, d, c, z	Solution of a distributed matrix equation, one matrix is triangular

[p?geadd](#)

Performs sum operation for two distributed general matrices.

Syntax

```
void psgeadd (const char *trans , const MKL_INT *m , const MKL_INT *n , const float
*alpha , const float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca , const float *beta , float *c , const MKL_INT *ic , const MKL_INT *jc , const
MKL_INT *descc );

void pdgeadd (const char *trans , const MKL_INT *m , const MKL_INT *n , const double
*alpha , const double *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT
*desca , const double *beta , double *c , const MKL_INT *ic , const MKL_INT *jc , const
MKL_INT *descc );

void pcgeadd (const char *trans , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const MKL_Complex8 *beta , MKL_Complex8 *c , const MKL_INT *ic ,
const MKL_INT *jc , const MKL_INT *descc );

void pzgeadd (const char *trans , const MKL_INT *m , const MKL_INT *n , const
MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const MKL_Complex16 *beta , MKL_Complex16 *c , const MKL_INT
*ic , const MKL_INT *jc , const MKL_INT *descc );
```

Include Files

- `mkc_pblas.h`

Description

The `p?geadd` routines perform sum operation for two distributed general matrices. The operation is defined as

$$\text{sub}(C) := \text{beta} * \text{sub}(C) + \text{alpha} * \text{op}(\text{sub}(A)),$$

where:

$\text{op}(x)$ is one of $\text{op}(x) = x$, or $\text{op}(x) = x'$,

α and β are scalars,

$\text{sub}(C)$ is an m -by- n distributed matrix, $\text{sub}(C) = C(\text{ic}:\text{ic}+m-1, \text{jc}:\text{jc}+n-1)$.

$\text{sub}(A)$ is a distributed matrix, $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+m-1)$.

Input Parameters

trans (global) Specifies the operation:
 if *trans* = 'N' or 'n', then $\text{op}(\text{sub}(A)) := \text{sub}(A)$;
 if *trans* = 'T' or 't', then $\text{op}(\text{sub}(A)) := \text{sub}(A)'$;

	if <i>trans</i> = 'C' or 'c', then $\text{op}(\text{sub}(A)) := \text{sub}(A)'$.
<i>m</i>	(global) Specifies the number of rows of the distributed matrix <i>sub(C)</i> and the number of columns of the submatrix <i>sub(A)</i> , $m \geq 0$.
<i>n</i>	(global) Specifies the number of columns of the distributed matrix <i>sub(C)</i> and the number of rows of the submatrix <i>sub(A)</i> , $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size (<i>lld_a</i> , $\text{LOCq}(ja+m-1)$). This array contains the local pieces of the distributed matrix <i>sub(A)</i> .
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then <i>sub(C)</i> need not be set on input.
<i>c</i>	(local) Array, size (<i>lld_c</i> , $\text{LOCq}(jc+n-1)$). This array contains the local pieces of the distributed matrix <i>sub(C)</i> .
<i>ic, jc</i>	(global) The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <i>sub(C)</i> , respectively.
<i>descc</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the updated submatrix.
----------	---------------------------------------

p?tradd

Performs sum operation for two distributed triangular matrices.

Syntax

```
void pstradd (const char *uplo , const char *trans , const MKL_INT *m , const MKL_INT
*n , const float *alpha , const float *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const float *beta , float *c , const MKL_INT *ic , const MKL_INT
*jc , const MKL_INT *descc );

void pdtradd (const char *uplo , const char *trans , const MKL_INT *m , const MKL_INT
*n , const double *alpha , const double *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const double *beta , double *c , const MKL_INT *ic , const
MKL_INT *jc , const MKL_INT *descc );
```

```

void pctradd (const char *uplo , const char *trans , const MKL_INT *m , const MKL_INT
*n , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *beta , MKL_Complex8 *c , const
MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );

void pztradd (const char *uplo , const char *trans , const MKL_INT *m , const MKL_INT
*n , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *beta , MKL_Complex16 *c ,
const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );

```

Include Files

- mkl_pblas.h

Description

The `p?tradd` routines perform sum operation for two distributed triangular matrices. The operation is defined as

```
sub(C) := beta*sub(C) + alpha*op(sub(A)),
```

where:

`op(x)` is one of `op(x) = x`, or `op(x) = x'`, or `op(x) = conjg(x')`.

alpha and *beta* are scalars,

`sub(C)` is an *m*-by-*n* distributed matrix, `sub(C)=C(ic:ic+m-1, jc:jc+n-1)`.

`sub(A)` is a distributed matrix, `sub(A)=A(ia:ia+n-1, ja:ja+m-1)`.

Input Parameters

<i>uplo</i>	(global) Specifies whether the distributed matrix <code>sub(C)</code> is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	(global) Specifies the operation: if <i>trans</i> = 'N' or 'n', then <code>op(sub(A)) := sub(A)</code> ; if <i>trans</i> = 'T' or 't', then <code>op(sub(A)) := sub(A)'</code> ; if <i>trans</i> = 'C' or 'c', then <code>op(sub(A)) := conjg(sub(A)')</code> .
<i>m</i>	(global) Specifies the number of rows of the distributed matrix <code>sub(C)</code> and the number of columns of the submatrix <code>sub(A)</code> , $m \geq 0$.
<i>n</i>	(global) Specifies the number of columns of the distributed matrix <code>sub(C)</code> and the number of rows of the submatrix <code>sub(A)</code> , $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size <code>(lld_a, LOCq(ja+m-1))</code> . This array contains the local pieces of the distributed matrix <code>sub(A)</code> .

<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then <i>sub(C)</i> need not be set on input.
<i>c</i>	(local) Array, size <code>(lld_c, LOCq(jc+n-1))</code> . This array contains the local pieces of the distributed matrix <i>sub(C)</i> .
<i>ic, jc</i>	(global) The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <i>sub(C)</i> , respectively.
<i>descb</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the updated submatrix.
----------	---------------------------------------

p?gemm

Computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product for distributed matrices.

Syntax

```
void psgemm (const char *transa , const char *transb , const MKL_INT *m , const MKL_INT
*n , const MKL_INT *k , const float *alpha , const float *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const float *b , const MKL_INT *ib , const MKL_INT
*jb , const MKL_INT *descb , const float *beta , float *c , const MKL_INT *ic , const
MKL_INT *jc , const MKL_INT *descb );
```

```
void pdgemm (const char *transa , const char *transb , const MKL_INT *m , const MKL_INT
*n , const MKL_INT *k , const double *alpha , const double *a , const MKL_INT *ia ,
const MKL_INT *ja , const MKL_INT *desca , const double *b , const MKL_INT *ib , const
MKL_INT *jb , const MKL_INT *descb , const double *beta , double *c , const MKL_INT
*ic , const MKL_INT *jc , const MKL_INT *descb );
```

```
void pcgemm (const char *transa , const char *transb , const MKL_INT *m , const MKL_INT
*n , const MKL_INT *k , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const
MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *b , const
MKL_INT *ib , const MKL_INT *jb , const MKL_INT *descb , const MKL_Complex8 *beta ,
MKL_Complex8 *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descb );
```

```
void pzgemm (const char *transa , const char *transb , const MKL_INT *m , const MKL_INT
*n , const MKL_INT *k , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const
MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *b , const
MKL_INT *ib , const MKL_INT *jb , const MKL_INT *descb , const MKL_Complex16 *beta ,
MKL_Complex16 *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descb );
```

Include Files

- `mkl_pblas.h`

Description

The `p?gemm` routines perform a matrix-matrix operation with general distributed matrices. The operation is defined as

```
sub(C) := alpha*op(sub(A))*op(sub(B)) + beta*sub(C),
```

where:

`op(x)` is one of `op(x) = x`, or `op(x) = x'`,

alpha and *beta* are scalars,

`sub(A)=A(ia:ia+m-1, ja:ja+k-1)`, `sub(B)=B(ib:ib+k-1, jb:jb+n-1)`, and `sub(C)=C(ic:ic+m-1, jc:jc+n-1)`, are distributed matrices.

Input Parameters

<i>transa</i>	(global) Specifies the form of <code>op(sub(A))</code> used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then <code>op(sub(A)) = sub(A)</code> ; if <i>transa</i> = 'T' or 't', then <code>op(sub(A)) = sub(A)'</code> ; if <i>transa</i> = 'C' or 'c', then <code>op(sub(A)) = sub(A)'</code> .
<i>transb</i>	(global) Specifies the form of <code>op(sub(B))</code> used in the matrix multiplication: if <i>transb</i> = 'N' or 'n', then <code>op(sub(B)) = sub(B)</code> ; if <i>transb</i> = 'T' or 't', then <code>op(sub(B)) = sub(B)'</code> ; if <i>transb</i> = 'C' or 'c', then <code>op(sub(B)) = sub(B)'</code> .
<i>m</i>	(global) Specifies the number of rows of the distributed matrices <code>op(sub(A))</code> and <code>sub(C)</code> , $m \geq 0$.
<i>n</i>	(global) Specifies the number of columns of the distributed matrices <code>op(sub(B))</code> and <code>sub(C)</code> , $n \geq 0$. The value of <i>n</i> must be at least zero.
<i>k</i>	(global) Specifies the number of columns of the distributed matrix <code>op(sub(A))</code> and the number of rows of the distributed matrix <code>op(sub(B))</code> . The value of <i>k</i> must be greater than or equal to 0.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> . When <i>alpha</i> is equal to zero, then the local entries of the arrays <i>a</i> and <i>b</i> corresponding to the entries of the submatrices <code>sub(A)</code> and <code>sub(B)</code> respectively need not be set on input.
<i>a</i>	(local) Array, size <code>lld_a</code> by <code>kla</code> , where <code>kla</code> is <code>LOCc(ja+k-1)</code> when <i>transa</i> = 'N' or 'n', and is <code>LOCq(ja+m-1)</code> otherwise. Before entry this array must contain the local pieces of the distributed matrix <code>sub(A)</code> .

<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>b</i>	(local) Array, size <code>lld_b</code> by <code>klb</code> , where <code>klb</code> is <code>LOCc(jb+n-1)</code> when <code>transb = 'N'</code> or <code>'n'</code> , and is <code>LOCq(jb+k-1)</code> otherwise. Before entry this array must contain the local pieces of the distributed matrix <code>sub(B)</code> .
<i>ib, jb</i>	(global) The row and column indices in the distributed matrix <i>B</i> indicating the first row and the first column of the submatrix <code>sub(B)</code> , respectively
<i>descb</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>B</i> .
<i>beta</i>	(global) Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then <code>sub(C)</code> need not be set on input.
<i>c</i>	(local) Array, size <code>(lld_a, LOCq(jc+n-1))</code> . Before entry this array must contain the local pieces of the distributed matrix <code>sub(C)</code> .
<i>ic, jc</i>	(global) The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <code>sub(C)</code> , respectively
<i>descc</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the <i>m</i> -by- <i>n</i> distributed matrix $\alpha * \text{op}(\text{sub}(A)) * \text{op}(\text{sub}(B)) + \text{beta} * \text{sub}(C)$.
----------	--

p?hemm

Performs a scalar-matrix-matrix product (one matrix operand is Hermitian) and adds the result to a scalar-matrix product.

Syntax

```
void pchemm (const char *side , const char *uplo , const MKL_INT *m , const MKL_INT
*n , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *b , const MKL_INT *ib , const
MKL_INT *jb , const MKL_INT *descb , const MKL_Complex8 *beta , MKL_Complex8 *c , const
MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );

void pzhemm (const char *side , const char *uplo , const MKL_INT *m , const MKL_INT
*n , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *b , const MKL_INT *ib , const
MKL_INT *jb , const MKL_INT *descb , const MKL_Complex16 *beta , MKL_Complex16 *c ,
const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
```


Include Files

- `mkl_pblas.h`

Description

The `p?hemm` routines perform a matrix-matrix operation with distributed matrices. The operation is defined as

```
sub(C) := alpha*sub(A)*sub(B) + beta*sub(C),
```

or

```
sub(C) := alpha*sub(B)*sub(A) + beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(A) is a Hermitian distributed matrix, *sub(A)* = *A*(*ia:ia+m-1*, *ja:ja+m-1*), if *side* = 'L', and *sub(A)* = *A*(*ia:ia+n-1*, *ja:ja+n-1*), if *side* = 'R'.

sub(B) and *sub(C)* are *m*-by-*n* distributed matrices.

sub(B) = *B*(*ib:ib+m-1*, *jb:jb+n-1*), *sub(C)* = *C*(*ic:ic+m-1*, *jc:jc+n-1*).

Input Parameters

<i>side</i>	<p>(global) Specifies whether the Hermitian distributed matrix <i>sub(A)</i> appears on the left or right in the operation:</p> <p>if <i>side</i> = 'L' or 'l', then <i>sub(C)</i> := <i>alpha</i>*<i>sub(A)</i> *<i>sub(B)</i> + <i>beta</i>*<i>sub(C)</i>;</p> <p>if <i>side</i> = 'R' or 'r', then <i>sub(C)</i> := <i>alpha</i>*<i>sub(B)</i> *<i>sub(A)</i> + <i>beta</i>*<i>sub(C)</i>.</p>
<i>uplo</i>	<p>(global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(A)</i> is used:</p> <p>if <i>uplo</i> = 'U' or 'u', then the upper triangular part is used;</p> <p>if <i>uplo</i> = 'L' or 'l', then the lower triangular part is used.</p>
<i>m</i>	<p>(global) Specifies the number of rows of the distribute submatrix <i>sub(C)</i>, $m \geq 0$.</p>
<i>n</i>	<p>(global) Specifies the number of columns of the distribute submatrix <i>sub(C)</i>, $n \geq 0$.</p>
<i>alpha</i>	<p>(global)</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>a</i>	<p>(local)</p> <p>Array, size (<i>lld_a</i>, <i>LOCq(ja+na-1)</i>).</p> <p>Before entry this array must contain the local pieces of the symmetric distributed matrix <i>sub(A)</i>, such that when <i>uplo</i> = 'U' or 'u', the <i>na</i>-by-<i>na</i> upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of <i>sub(A)</i> is not referenced, and when <i>uplo</i> = 'L' or</p>

'1', the na -by- na lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

ia, ja

(global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively

desca

(global and local) array of dimension 9. The array descriptor of the distributed matrix A .

b

(local)

Array, size $(lld_b, LOCq(jb+n-1))$. Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(B)$.

ib, jb

(global) The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.

descb

(global and local) array of dimension 9. The array descriptor of the distributed matrix B .

beta

(global)

Specifies the scalar *beta*.

When *beta* is set to zero, then $\text{sub}(C)$ need not be set on input.

c

(local)

Array, size $(lld_c, LOCq(jc+n-1))$. Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(C)$.

ic, jc

(global) The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively

descc

(global and local) array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

c

Overwritten by the m -by- n updated distributed matrix.

p?herk

Performs a rank- k update of a distributed Hermitian matrix.

Syntax

```
void pcherk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const float *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT
*ja , const MKL_INT *desca , const float *beta , MKL_Complex8 *c , const MKL_INT *ic ,
const MKL_INT *jc , const MKL_INT *descc );
```

```
void pzherk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const double *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT
*ja , const MKL_INT *desca , const double *beta , MKL_Complex16 *c , const MKL_INT
*ic , const MKL_INT *jc , const MKL_INT *descc );
```

Include Files

- `mkl_pblas.h`

Description

The `p?herk` routines perform a distributed matrix-matrix operation defined as

```
sub(C) := alpha*sub(A)*conjg(sub(A)') + beta*sub(C),
```

or

```
sub(C) := alpha*conjg(sub(A)')*sub(A) + beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(C) is an *n*-by-*n* Hermitian distributed matrix, *sub(C)* = *C*(*ic:ic+n-1*, *jc:jc+n-1*).

sub(A) is a distributed matrix, *sub(A)* = *A*(*ia:ia+n-1*, *ja:ja+k-1*), if *trans* = 'N' or 'n', and *sub(A)* = *A*(*ia:ia+k-1*, *ja:ja+n-1*) otherwise.

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(C)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(C)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(C)</i> is used.
<i>trans</i>	(global) Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>sub(C)</i> := <i>alpha</i> * <i>sub(A)</i> * <i>conjg(sub(A)')</i> + <i>beta</i> * <i>sub(C)</i> ; if <i>trans</i> = 'C' or 'c', then <i>sub(C)</i> := <i>alpha</i> * <i>conjg(sub(A)')</i> * <i>sub(A)</i> + <i>beta</i> * <i>sub(C)</i> .
<i>n</i>	(global) Specifies the order of the distributed matrix <i>sub(C)</i> , $n \geq 0$.
<i>k</i>	(global) On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the distributed matrix <i>sub(A)</i> , and on entry with <i>trans</i> = 'T' or 't' or 'C' or 'c', <i>k</i> specifies the number of rows of the distributed matrix <i>sub(A)</i> , $k \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size (<i>lld_a</i> , <i>kla</i>), where <i>kla</i> is <i>LOCq(ja+k-1)</i> when <i>trans</i> = 'N' or 'n', and is <i>LOCq(ja+n-1)</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', this array contains the local pieces of the distributed matrix <i>sub(A)</i> .
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .

<i>beta</i>	(global) Specifies the scalar <i>beta</i> .
<i>c</i>	(local) Array, size <code>(lld_c, LOCq(jc+n-1))</code> . Before entry with <i>uplo</i> = 'U' or 'u', this array contains <i>n</i> -by- <i>n</i> upper triangular part of the symmetric distributed matrix <code>sub(C)</code> and its strictly lower triangular part is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', this array contains <i>n</i> -by- <i>n</i> lower triangular part of the symmetric distributed matrix <code>sub(C)</code> and its strictly upper triangular part is not referenced.
<i>ic, jc</i>	(global) The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <code>sub(C)</code> , respectively.
<i>descc</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of <code>sub(C)</code> is overwritten by the upper triangular part of the updated distributed matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of <code>sub(C)</code> is overwritten by the upper triangular part of the updated distributed matrix.
----------	--

p?her2k

Performs a rank-2k update of a Hermitian distributed matrix.

Syntax

```
void pcher2k (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *b , const MKL_INT *ib , const
MKL_INT *jb , const MKL_INT *descb , const float *beta , MKL_Complex8 *c , const
MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );

void pzher2k (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *b , const MKL_INT *ib , const
MKL_INT *jb , const MKL_INT *descb , const double *beta , MKL_Complex16 *c , const
MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
```

Include Files

- `mk1_pblas.h`

Description

The `p?her2k` routines perform a distributed matrix-matrix operation defined as

```
sub(C) := alpha * sub(A) * conjg(sub(B)') + conjg(alpha) * sub(B) * conjg(sub(A)') + beta * sub(C) ,
```

or

```
sub(C) := alpha * conjg(sub(A)') * sub(A) + conjg(alpha) * conjg(sub(B)') * sub(B) + beta * sub(C) ,
```

where:

alpha and *beta* are scalars,

sub(C) is an *n*-by-*n* Hermitian distributed matrix, $\text{sub}(C) = C(ic:ic+n-1, jc:jc+n-1)$.

sub(A) is a distributed matrix, $\text{sub}(A) = A(ia:ia+n-1, ja:ja+k-1)$, if *trans* = 'N' or 'n', and $\text{sub}(A) = A(ia:ia+k-1, ja:ja+n-1)$ otherwise.

sub(B) is a distributed matrix, $\text{sub}(B) = B(ib:ib+n-1, jb:jb+k-1)$, if *trans* = 'N' or 'n', and $\text{sub}(B) = B(ib:ib+k-1, jb:jb+n-1)$ otherwise.

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(C)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(C)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(C)</i> is used.
<i>trans</i>	(global) Specifies the operation: if <i>trans</i> = 'N' or 'n', then $\text{sub}(C) := \alpha * \text{sub}(A) * \text{conjg}(\text{sub}(B)') + \text{conjg}(\alpha) * \text{sub}(B) * \text{conjg}(\text{sub}(A)') + \beta * \text{sub}(C)$; if <i>trans</i> = 'C' or 'c', then $\text{sub}(C) := \alpha * \text{conjg}(\text{sub}(A)') * \text{sub}(A) + \text{conjg}(\alpha) * \text{conjg}(\text{sub}(B)') * \text{sub}(B) + \beta * \text{sub}(C)$.
<i>n</i>	(global) Specifies the order of the distributed matrix <i>sub(C)</i> , $n \geq 0$.
<i>k</i>	(global) On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the distributed matrices <i>sub(A)</i> and <i>sub(B)</i> , and on entry with <i>trans</i> = 'C' or 'c', <i>k</i> specifies the number of rows of the distributed matrices <i>sub(A)</i> and <i>sub(B)</i> , $k \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size (<i>lld_a</i> , <i>kla</i>), where <i>kla</i> is $\text{LOCq}(ja+k-1)$ when <i>trans</i> = 'N' or 'n', and is $\text{LOCq}(ja+n-1)$ otherwise. Before entry with <i>trans</i> = 'N' or 'n', this array contains the local pieces of the distributed matrix <i>sub(A)</i> .
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>b</i>	(local) Array, size (<i>lld_b</i> , <i>kib</i>), where <i>kib</i> is $\text{LOCq}(jb+k-1)$ when <i>trans</i> = 'N' or 'n', and is $\text{LOCq}(jb+n-1)$ otherwise. Before entry with <i>trans</i> = 'N' or 'n', this array contains the local pieces of the distributed matrix <i>sub(B)</i> .
<i>ib, jb</i>	(global) The row and column indices in the distributed matrix <i>B</i> indicating the first row and the first column of the submatrix <i>sub(B)</i> , respectively.

<i>descb</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>B</i> .
<i>beta</i>	(global) Specifies the scalar <i>beta</i> .
<i>c</i>	(local) Array, size <code>(lld_c, LOCq(jc+n-1))</code> . Before entry with <i>uplo</i> = 'U' or 'u', this array contains <i>n</i> -by- <i>n</i> upper triangular part of the symmetric distributed matrix <code>sub(C)</code> and its strictly lower triangular part is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', this array contains <i>n</i> -by- <i>n</i> lower triangular part of the symmetric distributed matrix <code>sub(C)</code> and its strictly upper triangular part is not referenced.
<i>ic, jc</i>	(global) The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <code>sub(C)</code> , respectively.
<i>descc</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of <code>sub(C)</code> is overwritten by the upper triangular part of the updated distributed matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of <code>sub(C)</code> is overwritten by the upper triangular part of the updated distributed matrix.
----------	--

p?symm

Performs a scalar-matrix-matrix product (one matrix operand is symmetric) and adds the result to a scalar-matrix product for distribute matrices.

Syntax

```
void pssymm (const char *side , const char *uplo , const MKL_INT *m , const MKL_INT
*n , const float *alpha , const float *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const float *b , const MKL_INT *ib , const MKL_INT *jb , const
MKL_INT *descb , const float *beta , float *c , const MKL_INT *ic , const MKL_INT *jc ,
const MKL_INT *descc );

void pdsymm (const char *side , const char *uplo , const MKL_INT *m , const MKL_INT
*n , const double *alpha , const double *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const double *b , const MKL_INT *ib , const MKL_INT *jb , const
MKL_INT *descb , const double *beta , double *c , const MKL_INT *ic , const MKL_INT
*jc , const MKL_INT *descc );

void pcsymm (const char *side , const char *uplo , const MKL_INT *m , const MKL_INT
*n , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *b , const MKL_INT *ib , const
MKL_INT *jb , const MKL_INT *descb , const MKL_Complex8 *beta , MKL_Complex8 *c , const
MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
```

```
void pzsymm (const char *side , const char *uplo , const MKL_INT *m , const MKL_INT
*n , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *b , const MKL_INT *ib , const
MKL_INT *jb , const MKL_INT *descb , const MKL_Complex16 *beta , MKL_Complex16 *c ,
const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
```

Include Files

- mkl_pblas.h

Description

The `pzsymm` routines perform a matrix-matrix operation with distributed matrices. The operation is defined as

```
sub(C) := alpha*sub(A)*sub(B) + beta*sub(C) ,
```

or

```
sub(C) := alpha*sub(B)*sub(A) + beta*sub(C) ,
```

where:

alpha and *beta* are scalars,

sub(A) is a symmetric distributed matrix, *sub(A)* = *A*(*ia:ia+m-1*, *ja:ja+m-1*), if *side* = 'L', and
sub(A) = *A*(*ia:ia+n-1*, *ja:ja+n-1*), if *side* = 'R'.

sub(B) and *sub(C)* are *m*-by-*n* distributed matrices.

sub(B) = *B*(*ib:ib+m-1*, *jb:jb+n-1*), *sub(C)* = *C*(*ic:ic+m-1*, *jc:jc+n-1*).

Input Parameters

<i>side</i>	(global) Specifies whether the symmetric distributed matrix <i>sub(A)</i> appears on the left or right in the operation: if <i>side</i> = 'L' or 'l', then <i>sub(C)</i> := <i>alpha</i> * <i>sub(A)</i> * <i>sub(B)</i> + <i>beta</i> * <i>sub(C)</i> ; if <i>side</i> = 'R' or 'r', then <i>sub(C)</i> := <i>alpha</i> * <i>sub(B)</i> * <i>sub(A)</i> + <i>beta</i> * <i>sub(C)</i> .
<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(A)</i> is used: if <i>uplo</i> = 'U' or 'u', then the upper triangular part is used; if <i>uplo</i> = 'L' or 'l', then the lower triangular part is used.
<i>m</i>	(global) Specifies the number of rows of the distribute submatrix <i>sub(C)</i> , $m \geq 0$.
<i>n</i>	(global) Specifies the number of columns of the distribute submatrix <i>sub(C)</i> , $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size (<i>lld_a</i> , <i>LOCq(ja+na-1)</i>).

Before entry this array must contain the local pieces of the symmetric distributed matrix `sub(A)`, such that when `uplo = 'U' or 'u'`, the *na*-by-*na* upper triangular part of the distributed matrix `sub(A)` must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of `sub(A)` is not referenced, and when `uplo = 'L' or 'l'`, the *na*-by-*na* lower triangular part of the distributed matrix `sub(A)` must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of `sub(A)` is not referenced.

<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>b</i>	(local) Array, size <code>(lld_b, LOCq(jb+n-1))</code> . Before entry this array must contain the local pieces of the distributed matrix <code>sub(B)</code> .
<i>ib, jb</i>	(global) The row and column indices in the distributed matrix <i>B</i> indicating the first row and the first column of the submatrix <code>sub(B)</code> , respectively.
<i>descb</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>B</i> .
<i>beta</i>	(global) Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <code>sub(C)</code> need not be set on input.
<i>c</i>	(local) Array, size <code>(lld_c, LOCq(jc+n-1))</code> . Before entry this array must contain the local pieces of the distributed matrix <code>sub(C)</code> .
<i>ic, jc</i>	(global) The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <code>sub(C)</code> , respectively.
<i>descc</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the <i>m</i> -by- <i>n</i> updated matrix.
----------	---

p?syrk

Performs a rank-k update of a symmetric distributed matrix.

Syntax

```
void pssyrk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const float *alpha , const float *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const float *beta , float *c , const MKL_INT *ic , const MKL_INT
*jc , const MKL_INT *descc );
```



```

void pdsyrk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const double *alpha , const double *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const double *beta , double *c , const MKL_INT *ic , const
MKL_INT *jc , const MKL_INT *descc );

void pcsyrk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *beta , MKL_Complex8 *c , const
MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );

void pzsyrk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *beta , MKL_Complex16 *c ,
const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );

```

Include Files

- mkl_pblas.h

Description

The `p?syrk` routines perform a distributed matrix-matrix operation defined as

```
sub(C) := alpha*sub(A)*sub(A)' + beta*sub(C),
```

or

```
sub(C) := alpha*sub(A)'*sub(A) + beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(C) is an *n*-by-*n* symmetric distributed matrix, *sub(C)* = *C*(*ic:ic+n-1*, *jc:jc+n-1*).

sub(A) is a distributed matrix, *sub(A)* = *A*(*ia:ia+n-1*, *ja:ja+k-1*), if *trans* = 'N' or 'n', and *sub(A)* = *A*(*ia:ia+k-1*, *ja:ja+n-1*) otherwise.

Input Parameters

<i>uplo</i>	<p>(global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(C)</i> is used:</p> <p>If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(C)</i> is used.</p> <p>If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(C)</i> is used.</p>
<i>trans</i>	<p>(global) Specifies the operation:</p> <p>if <i>trans</i> = 'N' or 'n', then <i>sub(C)</i> := <i>alpha</i>*<i>sub(A)</i>*<i>sub(A)'</i> + <i>beta</i>*<i>sub(C)</i>;</p> <p>if <i>trans</i> = 'T' or 't', then <i>sub(C)</i> := <i>alpha</i>*<i>sub(A)'</i>*<i>sub(A)</i> + <i>beta</i>*<i>sub(C)</i>.</p>
<i>n</i>	(global) Specifies the order of the distributed matrix <i>sub(C)</i> , $n \geq 0$.
<i>k</i>	<p>(global) On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the distributed matrix <i>sub(A)</i>, and on entry with <i>trans</i> = 'T' or 't', <i>k</i> specifies the number of rows of the distributed matrix <i>sub(A)</i>, $k \geq 0$.</p>

<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size (<i>lld_a</i> , <i>kla</i>), where <i>kla</i> is <code>LOCq(ja+k-1)</code> when <i>trans</i> = 'N' or 'n', and is <code>LOCq(ja+n-1)</code> otherwise. Before entry with <i>trans</i> = 'N' or 'n', this array contains the local pieces of the distributed matrix <code>sub(A)</code> .
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) Specifies the scalar <i>beta</i> .
<i>c</i>	(local) Array, size (<i>lld_c</i> , <code>LOCq(jc+n-1)</code>). Before entry with <i>uplo</i> = 'U' or 'u', this array contains <i>n</i> -by- <i>n</i> upper triangular part of the symmetric distributed matrix <code>sub(C)</code> and its strictly lower triangular part is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', this array contains <i>n</i> -by- <i>n</i> lower triangular part of the symmetric distributed matrix <code>sub(C)</code> and its strictly upper triangular part is not referenced.
<i>ic, jc</i>	(global) The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <code>sub(C)</code> , respectively.
<i>descc</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of <code>sub(C)</code> is overwritten by the upper triangular part of the updated distributed matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of <code>sub(C)</code> is overwritten by the upper triangular part of the updated distributed matrix.
----------	--

p?syr2k

Performs a rank-2k update of a symmetric distributed matrix.

Syntax

```
void pssyr2k (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT *k , const float *alpha , const float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const float *b , const MKL_INT *ib , const MKL_INT *jb , const MKL_INT *descb , const float *beta , float *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
```

```

void pdsyr2k (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const double *alpha , const double *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_INT *desca , const double *b , const MKL_INT *ib , const MKL_INT *jb , const
MKL_INT *descb , const double *beta , double *c , const MKL_INT *ic , const MKL_INT
*jc , const MKL_INT *descc );

void pcsyr2k (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *b , const MKL_INT *ib , const
MKL_INT *jb , const MKL_INT *descb , const MKL_Complex8 *beta , MKL_Complex8 *c , const
MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );

void pzsyr2k (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT
*k , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const
MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *b , const MKL_INT *ib , const
MKL_INT *jb , const MKL_INT *descb , const MKL_Complex16 *beta , MKL_Complex16 *c ,
const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );

```

Include Files

- mkl_pblas.h

Description

The p?syr2k routines perform a distributed matrix-matrix operation defined as

$$\text{sub}(C) := \alpha \text{sub}(A) * \text{sub}(B) + \alpha \text{sub}(B) * \text{sub}(A) + \beta \text{sub}(C),$$

or

$$\text{sub}(C) := \alpha \text{sub}(A) * \text{sub}(B) + \alpha \text{sub}(B) * \text{sub}(A) + \beta \text{sub}(C),$$

where:

alpha and *beta* are scalars,

sub(C) is an *n*-by-*n* symmetric distributed matrix, *sub(C)* = *C*(*ic:ic+n-1, jc:jc+n-1*).

sub(A) is a distributed matrix, *sub(A)* = *A*(*ia:ia+n-1, ja:ja+k-1*), if *trans* = 'N' or 'n', and *sub(A)* = *A*(*ia:ia+k-1, ja:ja+n-1*) otherwise.

sub(B) is a distributed matrix, *sub(B)* = *B*(*ib:ib+n-1, jb:jb+k-1*), if *trans* = 'N' or 'n', and *sub(B)* = *B*(*ib:ib+k-1, jb:jb+n-1*) otherwise.

Input Parameters

<i>uplo</i>	(global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(C)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(C)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(C)</i> is used.
<i>trans</i>	(global) Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>sub(C) := alpha*sub(A)*sub(B) + alpha*sub(B)*sub(A) + beta*sub(C);</i> if <i>trans</i> = 'T' or 't', then <i>sub(C) := alpha*sub(B)*sub(A) + alpha*sub(A)*sub(B) + beta*sub(C).</i>

<i>n</i>	(global) Specifies the order of the distributed matrix $\text{sub}(C)$, $n \geq 0$.
<i>k</i>	(global) On entry with $\text{trans} = 'N'$ or $'n'$, <i>k</i> specifies the number of columns of the distributed matrices $\text{sub}(A)$ and $\text{sub}(B)$, and on entry with $\text{trans} = 'T'$ or $'t'$, <i>k</i> specifies the number of rows of the distributed matrices $\text{sub}(A)$ and $\text{sub}(B)$, $k \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size $(\text{lld_a}, \text{kla})$, where <i>kla</i> is $\text{LOCq}(ja+k-1)$ when $\text{trans} = 'N'$ or $'n'$, and is $\text{LOCq}(ja+n-1)$ otherwise. Before entry with $\text{trans} = 'N'$ or $'n'$, this array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>b</i>	(local) Array, size $(\text{lld_b}, \text{kib})$, where <i>kib</i> is $\text{LOCq}(jb+k-1)$ when $\text{trans} = 'N'$ or $'n'$, and is $\text{LOCq}(jb+n-1)$ otherwise. Before entry with $\text{trans} = 'N'$ or $'n'$, this array contains the local pieces of the distributed matrix $\text{sub}(B)$.
<i>ib, jb</i>	(global) The row and column indices in the distributed matrix <i>B</i> indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.
<i>descb</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>B</i> .
<i>beta</i>	(global) Specifies the scalar <i>beta</i> .
<i>c</i>	(local) Array, size $(\text{lld_c}, \text{LOCq}(jc+n-1))$. Before entry with $\text{uplo} = 'U'$ or $'u'$, this array contains <i>n</i> -by- <i>n</i> upper triangular part of the symmetric distributed matrix $\text{sub}(C)$ and its strictly lower triangular part is not referenced. Before entry with $\text{uplo} = 'L'$ or $'l'$, this array contains <i>n</i> -by- <i>n</i> lower triangular part of the symmetric distributed matrix $\text{sub}(C)$ and its strictly upper triangular part is not referenced.
<i>ic, jc</i>	(global) The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.
<i>descc</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	With $\text{uplo} = 'U'$ or $'u'$, the upper triangular part of $\text{sub}(C)$ is overwritten by the upper triangular part of the updated distributed matrix.
----------	---

With `uplo = 'L' or 'l'`, the lower triangular part of `sub(C)` is overwritten by the upper triangular part of the updated distributed matrix.

p?tran

Transposes a real distributed matrix.

Syntax

```
void pstran (const MKL_INT *m , const MKL_INT *n , const float *alpha , const float
*a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const float *beta ,
float *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );

void pdtran (const MKL_INT *m , const MKL_INT *n , const double *alpha , const double
*a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const double
*beta , double *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
```

Include Files

- `mkl_pblas.h`

Description

The `p?tran` routines transpose a real distributed matrix. The operation is defined as

$$\text{sub}(C) := \text{beta} * \text{sub}(C) + \text{alpha} * \text{sub}(A)^T,$$

where:

alpha and *beta* are scalars,

`sub(C)` is an *m*-by-*n* distributed matrix, `sub(C) = C(ic:ic+m-1, jc:jc+n-1)`.

`sub(A)` is a distributed matrix, `sub(A) = A(ia:ia+n-1, ja:ja+m-1)`.

Input Parameters

<i>m</i>	(global) Specifies the number of rows of the distributed matrix <code>sub(C)</code> , $m \geq 0$.
<i>n</i>	(global) Specifies the number of columns of the distributed matrix <code>sub(C)</code> , $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size <code>(lld_a, LOCq(ja+m-1))</code> . This array contains the local pieces of the distributed matrix <code>sub(A)</code> .
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) Specifies the scalar <i>beta</i> .

When *beta* is equal to zero, then `sub(C)` need not be set on input.

c

(local)

Array, size `(lld_c, LOCq(jc+n-1))`.

This array contains the local pieces of the distributed matrix `sub(C)`.

ic, jc

(global) The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix `sub(C)`, respectively.

descc

(global and local) array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c

Overwritten by the updated submatrix.

p?tranu

Transposes a distributed complex matrix.

Syntax

```
void pctransu (const MKL_INT *m , const MKL_INT *n , const MKL_Complex8 *alpha , const
MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const
MKL_Complex8 *beta , MKL_Complex8 *c , const MKL_INT *ic , const MKL_INT *jc , const
MKL_INT *descc );
```

```
void pztranu (const MKL_INT *m , const MKL_INT *n , const MKL_Complex16 *alpha , const
MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const
MKL_Complex16 *beta , MKL_Complex16 *c , const MKL_INT *ic , const MKL_INT *jc , const
MKL_INT *descc );
```

Include Files

- `mkc_pblas.h`

Description

The `p?tranu` routines transpose a complex distributed matrix. The operation is defined as

$$\text{sub}(C) := \text{beta} * \text{sub}(C) + \text{alpha} * \text{sub}(A)^T,$$

where:

alpha and *beta* are scalars,

`sub(C)` is an *m*-by-*n* distributed matrix, `sub(C) = C(ic:ic+m-1, jc:jc+n-1)`.

`sub(A)` is a distributed matrix, `sub(A) = A(ia:ia+n-1, ja:ja+m-1)`.

Input Parameters

m

(global) Specifies the number of rows of the distributed matrix `sub(C)`, $m \geq 0$.

n

(global) Specifies the number of columns of the distributed matrix `sub(C)`, $n \geq 0$.

alpha

(global)

	Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size $(l1d_a, LOCq(ja+m-1))$. This array contains the local pieces of the distributed matrix $sub(A)$.
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $sub(A)$, respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then $sub(C)$ need not be set on input.
<i>c</i>	(local) Array, size $(l1d_c, LOCq(jc+n-1))$. This array contains the local pieces of the distributed matrix $sub(C)$.
<i>ic, jc</i>	(global) The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix $sub(C)$, respectively.
<i>descc</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the updated submatrix.
----------	---------------------------------------

p?tranc

Transposes a complex distributed matrix, conjugated.

Syntax

```
void pctranc (const MKL_INT *m , const MKL_INT *n , const MKL_Complex8 *alpha , const
MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const
MKL_Complex8 *beta , MKL_Complex8 *c , const MKL_INT *ic , const MKL_INT *jc , const
MKL_INT *descc );
```

```
void pztranc (const MKL_INT *m , const MKL_INT *n , const MKL_Complex16 *alpha , const
MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const
MKL_Complex16 *beta , MKL_Complex16 *c , const MKL_INT *ic , const MKL_INT *jc , const
MKL_INT *descc );
```

Include Files

- `mkl_pblas.h`

Description

The `p?tranc` routines transpose a complex distributed matrix. The operation is defined as

$$sub(C) := beta * sub(C) + alpha * conjg(sub(A)'),$$

where:

alpha and *beta* are scalars,

$\text{sub}(C)$ is an m -by- n distributed matrix, $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$.

$\text{sub}(A)$ is a distributed matrix, $\text{sub}(A) = A(ia:ia+n-1, ja:ja+m-1)$.

Input Parameters

<i>m</i>	(global) Specifies the number of rows of the distributed matrix $\text{sub}(C)$, $m \geq 0$.
<i>n</i>	(global) Specifies the number of columns of the distributed matrix $\text{sub}(C)$, $n \geq 0$.
<i>alpha</i>	(global) Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) Array, size $(lld_a, LOCq(ja+m-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then $\text{sub}(C)$ need not be set on input.
<i>c</i>	(local) Array, size $(lld_c, LOCq(jc+n-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(C)$.
<i>ic, jc</i>	(global) The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.
<i>desc</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the updated submatrix.
----------	---------------------------------------

p?trmm

Computes a scalar-matrix-matrix product (one matrix operand is triangular) for distributed matrices.

Syntax

```
void pstrmm (const char *side , const char *uplo , const char *transa , const char
*diag , const MKL_INT *m , const MKL_INT *n , const float *alpha , const float *a ,
const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , float *b , const MKL_INT
*ib , const MKL_INT *jb , const MKL_INT *descb );
```



```

void pdtrmm (const char *side , const char *uplo , const char *transa , const char
*diag , const MKL_INT *m , const MKL_INT *n , const double *alpha , const double *a ,
const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , double *b , const
MKL_INT *ib , const MKL_INT *jb , const MKL_INT *descb );

void pctrmm (const char *side , const char *uplo , const char *transa , const char
*diag , const MKL_INT *m , const MKL_INT *n , const MKL_Complex8 *alpha , const
MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca ,
MKL_Complex8 *b , const MKL_INT *ib , const MKL_INT *jb , const MKL_INT *descb );

void pztrmm (const char *side , const char *uplo , const char *transa , const char
*diag , const MKL_INT *m , const MKL_INT *n , const MKL_Complex16 *alpha , const
MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca ,
MKL_Complex16 *b , const MKL_INT *ib , const MKL_INT *jb , const MKL_INT *descb );

```

Include Files

- mkl_pblas.h

Description

The `p?trmm` routines perform a matrix-matrix operation using triangular matrices. The operation is defined as

```
sub(B) := alpha*op(sub(A))*sub(B)
```

or

```
sub(B) := alpha*sub(B)*op(sub(A))
```

where:

alpha is a scalar,

sub(B) is an *m*-by-*n* distributed matrix, *sub(B)*=*B(ib:ib+m-1, jb:jb+n-1)*.

A is a unit, or non-unit, upper or lower triangular distributed matrix, *sub(A)*=*A(ia:ia+m-1, ja:ja+m-1)*, if *side* = 'L' or 'l', and *sub(A)*=*A(ia:ia+n-1, ja:ja+n-1)*, if *side* = 'R' or 'r'.

op(sub(A)) is one of *op(sub(A))* = *sub(A)*, or *op(sub(A))* = *sub(A)'*, or *op(sub(A))* = *conjg(sub(A)')*.

Input Parameters

<i>side</i>	(global) Specifies whether <i>op(sub(A))</i> appears on the left or right of <i>sub(B)</i> in the operation: if <i>side</i> = 'L' or 'l', then <i>sub(B) := alpha*op(sub(A))*sub(B)</i> ; if <i>side</i> = 'R' or 'r', then <i>sub(B) := alpha*sub(B)*op(sub(A))</i> .
<i>uplo</i>	(global) Specifies whether the distributed matrix <i>sub(A)</i> is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>transa</i>	(global) Specifies the form of <i>op(sub(A))</i> used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then <i>op(sub(A))</i> = <i>sub(A)</i> ; if <i>transa</i> = 'T' or 't', then <i>op(sub(A))</i> = <i>sub(A)'</i> ; if <i>transa</i> = 'C' or 'c', then <i>op(sub(A))</i> = <i>conjg(sub(A)')</i> .

<i>diag</i>	<p>(global) Specifies whether the matrix $\text{sub}(A)$ is unit triangular:</p> <p>if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular;</p> <p>if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.</p>
<i>m</i>	(global) Specifies the number of rows of the distributed matrix $\text{sub}(B)$, $m \geq 0$.
<i>n</i>	(global) Specifies the number of columns of the distributed matrix $\text{sub}(B)$, $n \geq 0$.
<i>alpha</i>	<p>(global)</p> <p>Specifies the scalar <i>alpha</i>.</p> <p>When <i>alpha</i> is zero, then the array <i>b</i> need not be set before entry.</p>
<i>a</i>	<p>(local)</p> <p>Array, size <i>lld_a</i> by <i>ka</i>, where <i>ka</i> is at least $\text{LOCq}(1, ja+m-1)$ when <i>side</i> = 'L' or 'l' and is at least $\text{LOCq}(1, ja+n-1)$ when <i>side</i> = 'R' or 'r'.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.</p> <p>When <i>diag</i> = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.</p>
<i>ia, ja</i>	(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>b</i>	<p>(local)</p> <p>Array, size $(\text{lld}_b, \text{LOCq}(1, jb+n-1))$.</p> <p>Before entry, this array contains the local pieces of the distributed matrix $\text{sub}(B)$.</p>
<i>ib, jb</i>	(global) The row and column indices in the distributed matrix <i>B</i> indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.
<i>descb</i>	(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>B</i> .

Output Parameters

<i>b</i>	Overwritten by the transformed distributed matrix.
----------	--

p?trsm

Solves a distributed matrix equation (one matrix operand is triangular).

Syntax

```
void pstrsm (const char *side , const char *uplo , const char *transa , const char
*diag , const MKL_INT *m , const MKL_INT *n , const float *alpha , const float *a ,
const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , float *b , const MKL_INT
*ib , const MKL_INT *jb , const MKL_INT *descb );
```

```
void pdtrsm (const char *side , const char *uplo , const char *transa , const char
*diag , const MKL_INT *m , const MKL_INT *n , const double *alpha , const double *a ,
const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , double *b , const
MKL_INT *ib , const MKL_INT *jb , const MKL_INT *descb );
```

```
void pctrsm (const char *side , const char *uplo , const char *transa , const char
*diag , const MKL_INT *m , const MKL_INT *n , const MKL_Complex8 *alpha , const
MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca ,
MKL_Complex8 *b , const MKL_INT *ib , const MKL_INT *jb , const MKL_INT *descb );
```

```
void pztrsm (const char *side , const char *uplo , const char *transa , const char
*diag , const MKL_INT *m , const MKL_INT *n , const MKL_Complex16 *alpha , const
MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca ,
MKL_Complex16 *b , const MKL_INT *ib , const MKL_INT *jb , const MKL_INT *descb );
```

Include Files

- mkl_pblas.h

Description

The p?trsm routines solve one of the following distributed matrix equations:

$$\text{op}(\text{sub}(A)) * X = \alpha * \text{sub}(B),$$

or

$$X * \text{op}(\text{sub}(A)) = \alpha * \text{sub}(B),$$

where:

α is a scalar,

X and $\text{sub}(B)$ are m -by- n distributed matrices, $\text{sub}(B) = B(ib:ib+m-1, jb:jb+n-1)$;

A is a unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(ia:ia+m-1, ja:ja+m-1)$, if $side = 'L'$ or $'l'$, and $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$, if $side = 'R'$ or $'r'$;

$\text{op}(\text{sub}(A))$ is one of $\text{op}(\text{sub}(A)) = \text{sub}(A)$, or $\text{op}(\text{sub}(A)) = \text{sub}(A)'$, or $\text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)')$.

The distributed matrix $\text{sub}(B)$ is overwritten by the solution matrix X .

Input Parameters

side (global) Specifies whether $\text{op}(\text{sub}(A))$ appears on the left or right of X in the equation:

if $side = 'L'$ or $'l'$, then $\text{op}(\text{sub}(A)) * X = \alpha * \text{sub}(B)$;

if $side = 'R'$ or $'r'$, then $X * \text{op}(\text{sub}(A)) = \alpha * \text{sub}(B)$.

<i>uplo</i>	<p>(global) Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular:</p> <p>if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular;</p> <p>if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.</p>
<i>transa</i>	<p>(global) Specifies the form of $\text{op}(\text{sub}(A))$ used in the matrix equation:</p> <p>if <i>transa</i> = 'N' or 'n', then $\text{op}(\text{sub}(A)) = \text{sub}(A)$;</p> <p>if <i>transa</i> = 'T' or 't', then $\text{op}(\text{sub}(A)) = \text{sub}(A)^T$;</p> <p>if <i>transa</i> = 'C' or 'c', then $\text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)^T)$.</p>
<i>diag</i>	<p>(global) Specifies whether the matrix $\text{sub}(A)$ is unit triangular:</p> <p>if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular;</p> <p>if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.</p>
<i>m</i>	<p>(global) Specifies the number of rows of the distributed matrix $\text{sub}(B)$, $m \geq 0$.</p>
<i>n</i>	<p>(global) Specifies the number of columns of the distributed matrix $\text{sub}(B)$, $n \geq 0$.</p>
<i>alpha</i>	<p>(global)</p> <p>Specifies the scalar <i>alpha</i>.</p> <p>When <i>alpha</i> is zero, then <i>a</i> is not referenced and <i>b</i> need not be set before entry.</p>
<i>a</i>	<p>(local)</p> <p>Array, size <i>lld_a</i> by <i>ka</i>, where <i>ka</i> is at least $\text{LOCq}(1, ja+m-1)$ when <i>side</i> = 'L' or 'l' and is at least $\text{LOCq}(1, ja+n-1)$ when <i>side</i> = 'R' or 'r'.</p> <p>Before entry with <i>uplo</i> = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced .</p> <p>When <i>diag</i> = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.</p>
<i>ia, ja</i>	<p>(global) The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.</p>
<i>desca</i>	<p>(global and local) array of dimension 9. The array descriptor of the distributed matrix <i>A</i>.</p>
<i>b</i>	<p>(local)</p> <p>Array, size (<i>lld_b</i>, $\text{LOCq}(1, jb+n-1)$).</p>

Before entry, this array contains the local pieces of the distributed matrix $\text{sub}(B)$.

ib, jb

(global) The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.

descb

(global and local) array of dimension 9. The array descriptor of the distributed matrix B .

Output Parameters

b

Overwritten by the solution distributed matrix X .

Partial Differential Equations Support

The Intel® oneAPI Math Kernel Library (oneMKL) provides tools for solving Partial Differential Equations (PDE). These tools are Trigonometric Transform interface routines (see [Trigonometric Transform Routines](#)) and Poisson Solver (see [Fast Poisson Solver Routines](#)).

Poisson Solver is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The solver is based on the Trigonometric Transform interface, which is, in turn, based on the Intel® oneAPI Math Kernel Library (oneMKL) Fast Fourier Transform (FFT) interface (refer to [Fourier Transform Functions](#)), optimized for Intel® processors.

Direct use of the Trigonometric Transform routines may be helpful to those who have already implemented their own solvers similar to the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver. As it may be hard enough to modify the original code so as to make it work with Poisson Solver, you are encouraged to use fast (staggered) sine/cosine transforms implemented in the Trigonometric Transform interface to improve performance of your solver.

Both Trigonometric Transform and Poisson Solver routines can be called from C and Fortran.

Product and Performance Information

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

Notice revision #20201201

Trigonometric Transform Routines

In addition to the Fast Fourier Transform (FFT) interface, described in [Fast Fourier Transforms](#), Intel® oneAPI Math Kernel Library (oneMKL) supports the Real Discrete Trigonometric Transforms (sometimes called real-to-real Discrete Fourier Transforms) interface. In this document, the interface is referred to as TT interface. It implements a group of routines (TT routines) used to compute sine/cosine, staggered sine/cosine, and twice staggered sine/cosine transforms (referred to as staggered2 sine/cosine transforms, for brevity). The TT interface provides much flexibility of use: you can adjust routines to your particular needs at the cost of manually tuning routine parameters or just call routines with default parameter values. The current Intel® oneAPI Math Kernel Library (oneMKL) implementation of the TT interface can be used in solving partial differential equations and contains routines that are helpful for Fast Poisson and similar solvers.

For the list of Trigonometric Transforms currently implemented in Intel® oneAPI Math Kernel Library (oneMKL) TT interface, see [Transforms Implemented](#).

If you have got used to the FFTW interface (www.fftw.org), you can call the TT interface functions through real-to-real FFTW to Intel® oneAPI Math Kernel Library (oneMKL) wrappers without changing FFTW function calls in your code (refer to [FFTW to Intel® MKL Wrappers for FFTW 3.x](#) for details). However, you are strongly encouraged to use the native TT interface for better performance. Another reason why you should use the

wrappers cautiously is that TT and the real-to-real FFTW interfaces are not fully compatible and some features of the real-to-real FFTW, such as strides and multidimensional transforms, are not available through wrappers.

Trigonometric Transforms Implemented

TT routines allow computing the following transforms:

Forward sine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^{n-1} f(i) \sin \frac{ki\pi}{n}, \quad k = 1, \dots, n-1$$

Backward sine transform

$$f(i) = \sum_{k=1}^{n-1} F(k) \sin \frac{ki\pi}{n}, \quad i = 1, \dots, n-1$$

Forward staggered sine transform

$$F(k) = \frac{1}{n} \sin \frac{(2k-1)\pi}{2} f(n) + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \sin \frac{(2k-1)i\pi}{2n}, \quad k = 1, \dots, n$$

Backward staggered sine transform

$$f(i) = \sum_{k=1}^n F(k) \sin \frac{(2k-1)i\pi}{2n}, \quad i = 1, \dots, n$$

Forward staggered2 sine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^n f(i) \sin \frac{(2k-1)(2i-1)\pi}{4n}, \quad k = 1, \dots, n$$

Backward staggered2 sine transform

$$f(i) = \sum_{k=1}^n F(k) \sin \frac{(2k-1)(2i-1)\pi}{4n}, \quad i = 1, \dots, n$$

Forward cosine transform

$$F(k) = \frac{1}{n} [f(0) + f(n) \cos k\pi] + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \cos \frac{ki\pi}{n}, \quad k = 0, \dots, n$$

Backward cosine transform

$$f(i) = \frac{1}{2} [F(0) + F(n) \cos i\pi] + \sum_{k=1}^{n-1} F(k) \cos \frac{ki\pi}{n}, \quad i = 0, \dots, n$$

Forward staggered cosine transform

$$F(k) = \frac{1}{n} f(0) + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \cos \frac{(2k+1)i\pi}{2n}, \quad k = 0, \dots, n-1$$

Backward staggered cosine transform

$$f(i) = \sum_{k=0}^{n-1} F(k) \cos \frac{(2k+1)i\pi}{2n}, \quad i = 0, \dots, n-1$$

Forward staggered2 cosine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^n f(i) \cos \frac{(2k-1)(2i-1)\pi}{4n}, \quad k = 1, \dots, n$$

Backward staggered2 cosine transform

$$f(i) = \sum_{k=1}^n F(k) \cos \frac{(2k-1)(2i-1)\pi}{4n}, i = 1, \dots, n$$

NOTE

The size of the transform n can be any integer greater or equal to 2.

Sequence of Invoking TT Routines

Computation of a transform using TT interface is conceptually divided into four steps, each of which is performed via a dedicated routine. [Table "TT Interface Routines"](#) lists the routines and briefly describes their purpose and use.

Most TT routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names.

TT Interface Routines

Routine	Description
<code>?_init_trig_transform</code>	Initializes basic data structures of Trigonometric Transforms.
<code>?_commit_trig_transform</code>	Checks consistency and correctness of user-defined data and creates a data structure to be used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface ¹ .
<code>?_forward_trig_transform</code> <code>?_backward_trig_transform</code>	Computes a forward/backward Trigonometric Transform of a specified type using the appropriate formula (see Transforms Implemented).
<code>free_trig_transform</code>	Releases the memory used by a data structure needed for calling FFT interface ¹ .

¹TT routines call Intel® oneAPI Math Kernel Library (oneMKL) FFT interface for better performance.

To find a transformed vector for a particular input vector only once, the Intel® oneAPI Math Kernel Library (oneMKL) TT interface routines are normally invoked in the order in which they are listed in [Table "TT Interface Routines"](#).

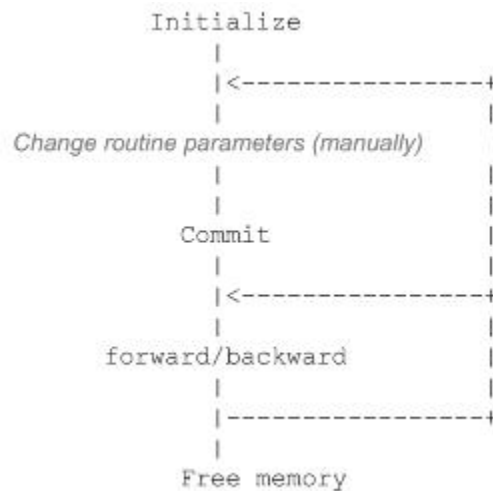
NOTE

Though the order of invoking TT routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in [Figure "Typical Order of Invoking TT Interface Routines"](#) indicates the typical order in which TT interface routines can be invoked in a general case (prefixes and suffixes in routine names are omitted).

`__border__top`

Typical Order of Invoking TT Interface Routines



A general scheme of using TT routines for double-precision computations is shown below. A similar scheme holds for single-precision computations with the only difference in the initial letter of routine names.

```

...
    d_init_trig_transform(&n, &tt_type, ipar, dpar, &ir);
/* Change parameters in ipar if necessary. */
/* Note that the result of the Transform will be in f. If you want to preserve the data stored
in f,
save it to another location before the function call below */
    d_commit_trig_transform(f, &handle, ipar, dpar, &ir);
    d_forward_trig_transform(f, &handle, ipar, dpar, &ir);
    d_backward_trig_transform(f, &handle, ipar, dpar, &ir);
    free_trig_transform(&handle, ipar, &ir);
/* here the user may clean the memory used by f, dpar, ipar */
...

```

You can find examples of code that uses TT interface routines to solve one-dimensional Helmholtz problem in the `examples\pdettt\source` folder in your Intel® oneAPI Math Kernel Library (oneMKL) directory.

Trigonometric Transform Interface Description

All types in this documentation are either standard C types `float` and `double` or `MKL_INT` integer type. For more information on the C types, refer to [C Datatypes Specific to oneMKL](#) and the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*. To better understand usage of the types, see examples in the `examples\pdettt\source` folder in your Intel® oneAPI Math Kernel Library (oneMKL) directory.

Routine Options

All TT routines use parameters to pass various options to one another. These parameters are arrays `ipar`, `dpar` and `spar`. Values for these parameters should be specified very carefully (see [Common Parameters](#)). You can change these values during computations to meet your needs.

WARNING

To avoid failure or incorrect results, you must provide correct and consistent parameters to the routines.

User Data Arrays

TT routines take arrays of user data as input. For example, user arrays are passed to the routine `d_forward_trig_transform` to compute a forward Trigonometric Transform. To minimize storage requirements and improve the overall run-time efficiency, Intel® oneAPI Math Kernel Library (oneMKL) TT routines do not make copies of user input arrays.

NOTE

If you need a copy of your input data arrays, you must save them yourself.

For better performance, align your data arrays as recommended in the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* (search the document for coding techniques to improve performance).

TT Routines

The section gives detailed description of TT routines, their syntax, parameters and values they return. Double-precision and single-precision versions of the same routine are described together.

TT routines call Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (described in [FFT Functions](#)), which enhances performance of the routines.

?_init_trig_transform

Initializes basic data structures of a Trigonometric Transform.

Syntax

```
void d_init_trig_transform(MKL_INT *n, MKL_INT *tt_type, MKL_INT ipar[], double dpar[],
MKL_INT *stat);

void s_init_trig_transform(MKL_INT *n, MKL_INT *tt_type, MKL_INT ipar[], float spar[],
MKL_INT *stat);
```

Include Files

- `mkl.h`

Input Parameters

<code>n</code>	MKL_INT*. Contains the size of the problem, which should be a positive integer greater than 1. Note that data vector of the transform, which other TT routines will use, must have size $n+1$ for all but staggered2 transforms. Staggered2 transforms require the vector of size n .
<code>tt_type</code>	MKL_INT*. Contains the type of transform to compute, defined via a set of named constants. The following constants are available in the current implementation of TT interface: MKL_SINE_TRANSFORM, MKL_STAGGERED_SINE_TRANSFORM, MKL_STAGGERED2_SINE_TRANSFORM; MKL_COSINE_TRANSFORM, MKL_STAGGERED_COSINE_TRANSFORM, MKL_STAGGERED2_COSINE_TRANSFORM.

Output Parameters

<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar</i> [6]. The status should be 0 to proceed to other TT routines.

Description

The `?_init_trig_transform` routine initializes basic data structures for Trigonometric Transforms of appropriate precision. After a call to `?_init_trig_transform`, all subsequently invoked TT routines use values of *ipar* and *dpar* (*spar*) array parameters returned by `?_init_trig_transform`. The routine initializes the entire array *ipar*. In the *dpar* or *spar* array, `?_init_trig_transform` initializes elements that do not depend upon the type of transform. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to [Common Parameters](#). You can skip a call to the initialization routine in your code. For more information, see [Caveat on Parameter Modifications](#).

Return Values

<i>stat</i> = 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this <i>stat</i> value.
<i>stat</i> = -99999	The routine failed to complete the task.

?_commit_trig_transform

Checks consistency and correctness of user's data as well as initializes certain data structures required to perform the Trigonometric Transform.

Syntax

```
void d_commit_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);

void s_commit_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- mkl.h

Input Parameters

<i>f</i>	double for <code>d_commit_trig_transform</code> , float for <code>s_commit_trig_transform</code> ,
----------	---

array of size n for staggered2 transforms and of size $n+1$ for all other transforms, where n is the size of the problem. Contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors:

- $f[0]$ and $f[n]$ for sine transforms
- $f[n]$ for staggered cosine transforms
- $f[0]$ for staggered sine transforms.

Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. These restrictions meet the requirements of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver, which the TT interface is primarily designed for (for details, see [Fast Poisson Solver Routines](#)).

<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.

Output Parameters

<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer to FFT Functions).
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar</i> [6] is updated with the <i>stat</i> value.
<i>dpar</i>	Contains double-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.
<i>spar</i>	Contains single-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar</i> [6].

Description

The routine `?_commit_trig_transform` checks consistency and correctness of the parameters to be passed to the transform routines `?_forward_trig_transform` and/or `?_backward_trig_transform`. The routine also initializes the following data structures: *handle*, *dpar* in case of `d_commit_trig_transform`, and *spar* in case of `s_commit_trig_transform`. The `?_commit_trig_transform` routine initializes only those elements of *dpar* or *spar* that depend upon the type of transform, defined in the `?_init_trig_transform` routine and passed to `?_commit_trig_transform` with the *ipar* array. The size of the problem n , which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called `?_init_trig_transform` routine. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to [Common Parameters](#). The routine performs only a basic check for correctness and consistency

of the parameters. If you are going to modify parameters of TT routines, see [Caveat on Parameter Modifications](#). Unlike `?_init_trig_transform`, you must call the `?_commit_trig_transform` routine in your code.

Return Values

`stat= 11`

The routine produced some warnings and made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning `ipar[6]=0` if you are sure that the parameters are correct.

`stat= 10`

The routine made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning `ipar[6]=0` if you are sure that the parameters are correct.

`stat= 1`

The routine produced some warnings. You may proceed with computations by assigning `ipar[6]=0` if you are sure that the parameters are correct.

`stat= 0`

The routine completed the task normally.

`stat= -100`

The routine stopped for any of the following reasons:

- An error in the user's data was encountered.
- Data in `ipar`, `dpar` or `spar` parameters became incorrect and/or inconsistent as a result of modifications.

`stat= -1000`

The routine stopped because of an FFT interface error.

`stat= -10000`

The routine stopped because the initialization failed to complete or the parameter `ipar[0]` was altered by mistake.

NOTE

Although positive values of `stat` usually indicate minor problems with the input data and Trigonometric Transform computations can be continued, you are highly recommended to investigate the problem first and achieve `stat=0`.

[?_forward_trig_transform](#)

Computes the forward Trigonometric Transform of type specified by the parameter.

Syntax

```
void d_forward_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
```

```
void s_forward_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- `mkh.h`

Input Parameters

<i>f</i>	<p>double for <code>d_forward_trig_transform</code>, float for <code>s_forward_trig_transform</code>,</p> <p>array of size n for staggered2 transforms and of size $n+1$ for all other transforms, where n is the size of the problem. On input, contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors:</p> <ul style="list-style-type: none"> • $f[0]$ and $f[n]$ for sine transforms • $f[n]$ for staggered cosine transforms • $f[0]$ for staggered sine transforms. <p>Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver, which the TT interface is primarily designed for (for details, see Fast Poisson Solver Routines).</p>
<i>handle</i>	<p>DEFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, see FFT Functions).</p>
<i>ipar</i>	<p>MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.</p>
<i>dpar</i>	<p>double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.</p>
<i>spar</i>	<p>float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.</p>

Output Parameters

<i>f</i>	Contains the transformed vector on output.
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar</i> [6] is updated with the <i>stat</i> value.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar</i> [6].

Description

The routine computes the forward Trigonometric Transform of type defined in the `?_init_trig_transform` routine and passed to `?_forward_trig_transform` with the *ipar* array. The size of the problem n , which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called `?_init_trig_transform` routine. The other data that facilitates the computation is created by `?_commit_trig_transform` and supplied in *dpar* or *spar*. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to [Common Parameters](#). The routine has a commit step, which calls the `?_commit_trig_transform` routine. The transform is computed according to formulas given in [Transforms Implemented](#). The routine replaces the input vector *f* with the transformed vector.

NOTE

If you need a copy of the data vector f to be transformed, make the copy before calling the `?_forward_trig_transform` routine.

Return Values

`stat= 0`

The routine completed the task normally.

`stat= -100`

The routine stopped for any of the following reasons:

- An error in the user's data was encountered.
- Data in `ipar`, `dpar` or `spar` parameters became incorrect and/or inconsistent as a result of modifications.

`stat= -1000`

The routine stopped because of an FFT interface error.

`stat= -10000`

The routine stopped because its commit step failed to complete or the parameter `ipar[0]` was altered by mistake.

?_backward_trig_transform

Computes the backward Trigonometric Transform of type specified by the parameter.

Syntax

```
void d_backward_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
```

```
void s_backward_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- `mkl.h`

Input Parameters

f

double for `d_backward_trig_transform`,

float for `s_backward_trig_transform`,

array of size n for staggered2 transforms and of size $n+1$ for all other transforms, where n is the size of the problem. On input, contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors:

- $f[0]$ and $f[n]$ for sine transforms
- $f[n]$ for staggered cosine transforms
- $f[0]$ for staggered sine transforms.

Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver, which the TT interface is primarily designed for (for details, see [Fast Poisson Solver Routines](#)).

<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, see FFT Functions).
<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

<i>f</i>	Contains the transformed vector on output.
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar</i> [6] is updated with the <i>stat</i> value.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar</i> [6].

Description

The routine computes the backward Trigonometric Transform of type defined in the `?_init_trig_transform` routine and passed to `?_backward_trig_transform` with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called `?_init_trig_transform` routine. The other data that facilitates the computation is created by `?_commit_trig_transform` and supplied in *dpar* or *spar*. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to [Common Parameters](#). The routine has a commit step, which calls the `?_commit_trig_transform` routine. The transform is computed according to formulas given in [Transforms Implemented](#). The routine replaces the input vector *f* with the transformed vector.

NOTE

If you need a copy of the data vector *f* to be transformed, make the copy before calling the `?_backward_trig_transform` routine.

Return Values

<i>stat</i> = 0	The routine completed the task normally.
<i>stat</i> = -100	The routine stopped for any of the following reasons: <ul style="list-style-type: none"> • An error in the user's data was encountered. • Data in <i>ipar</i>, <i>dpar</i> or <i>spar</i> parameters became incorrect and/or inconsistent as a result of modifications.
<i>stat</i> = -1000	The routine stopped because of an FFT interface error.
<i>stat</i> = -10000	The routine stopped because its commit step failed to complete or the parameter <i>ipar</i> [0] was altered by mistake.

free_trig_transform

Cleans the memory allocated for the data structure used by the FFT interface.

Syntax

```
void free_trig_transform(DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], MKL_INT *stat);
```

Include Files

- mkl.h

Input Parameters

<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer to FFT Functions).

Output Parameters

<i>handle</i>	The data structure used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface. Memory allocated for the structure is released on output.
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar[6]</i> is updated with the <i>stat</i> value.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar[6]</i> .

Description

The `free_trig_transform` routine cleans the memory used by the *handle* structure, needed for Intel® oneAPI Math Kernel Library (oneMKL) FFT functions. To release the memory allocated for other parameters, include cleaning of the memory in your code.

Return Values

<i>stat</i> = 0	The routine completed the task normally.
<i>stat</i> = -1000	The routine stopped because of an FFT interface error.
<i>stat</i> = -99999	The routine failed to complete the task.

Common Parameters of the Trigonometric Transforms

This section provides description of array parameters that hold TT routine options: *ipar*, *dpar* and *spar*.

NOTE

Initial values are assigned to the array parameters by the appropriate `?_init_trig_transform` and `?_commit_trig_transform` routines.

ipar MKL_INT array of size 128, holds integer data needed for Trigonometric Transform computations. Its elements are described in [Table "Elements of the ipar Array"](#):

Elements of the ipar Array

Index	Description
0	Contains the size of the problem to solve. The <code>?_init_trig_transform</code> routine sets <code>ipar[0]=n</code> , and all subsequently called TT routines use <code>ipar[0]</code> as the size of the transform.
1	<p>Contains error messaging options:</p> <ul style="list-style-type: none"> <code>ipar[1]=-1</code> indicates that all error messages will be printed to the file <code>MKL_Trig_Transforms_log.txt</code> in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. <code>ipar[1]=0</code> indicates that no error messages will be printed. <code>ipar[1]=1</code> (default) indicates that all error messages will be printed to the preconnected default output device (usually, screen). <p>In case of errors, each TT routine assigns a non-zero value to <code>stat</code> regardless of the <code>ipar[1]</code> setting.</p>
2	<p>Contains warning messaging options:</p> <ul style="list-style-type: none"> <code>ipar[2]=-1</code> indicates that all warning messages will be printed to the file <code>MKL_Trig_Transforms_log.txt</code> in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. <code>ipar[2]=0</code> indicates that no warning messages will be printed. <code>ipar[2]=1</code> (default) indicates that all warning messages will be printed to the preconnected default output device (usually, screen). <p>In case of warnings, the <code>stat</code> parameter will acquire a non-zero value regardless of the <code>ipar[2]</code> setting.</p>
3 through 4	Reserved for future use.
5	Contains the type of the transform. The <code>?_init_trig_transform</code> routine sets <code>ipar[5]=tt_type</code> , and all subsequently called TT routines use <code>ipar[5]</code> as the type of the transform.
6	Contains the <code>stat</code> value returned by the last completed TT routine. Used to check that the previous call to a TT routine completed with <code>stat=0</code> .
7	<p>Informs the <code>?_commit_trig_transform</code> routines whether to initialize data structures <code>dpar</code> (<code>spar</code>) and <code>handle</code>. <code>ipar[7]=0</code> indicates that the routine should skip the initialization and only check correctness and consistency of the parameters. Otherwise, the routine initializes the data structures. The default value is 1.</p> <p>The possibility to check correctness and consistency of input data without initializing data structures <code>dpar</code>, <code>spar</code> and <code>handle</code> enables avoiding performance losses in a repeated use of the same transform for different data vectors. Note that you can benefit from the opportunity that <code>ipar[7]</code> gives only if you are sure to have supplied proper tolerance value in the <code>dpar</code> or <code>spar</code> array. Otherwise, avoid tuning this parameter.</p>
8	Contains message style options for TT routines. Specifically, if <code>ipar[8]</code> is non-zero, TT routines print the messages in C-style notations. The default value is 1.

Index	Description
	When specifying message style options, be aware that by default, numbering of elements in C arrays starts at 0. For example, " <i>parameter ipar[0]=3 should be an even integer</i> " is a C-style message. The use of <i>ipar[8]</i> enables you to view messages in a more convenient style.
9	Specifies the number of OpenMP threads to run TT routines in the OpenMP environment of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver. The default value is 1. You are highly recommended not to alter this value. See also Caveat on Parameter Modifications .
10	Specifies the mode of compatibility with FFTW. The default value is 0. Set the value to 1 to invoke compatibility with FFTW. In the latter case, results will not be normalized, because FFTW does not do this. It is highly recommended not to alter this value, but rather use real-to-real FFTW to MKL wrappers, described in FFTW to Intel® MKL Wrappers for FFTW 3.x . See also Caveat on Parameter Modifications .
11 through 127	Reserved for future use.

NOTE

While you can declare the *ipar* array as `MKL_INT ipar[11]`, for future compatibility you should declare *ipar* as `MKL_INT ipar[128]`.

Arrays *dpar* and *spar* are the same except in the data precision:

<i>dpar</i>	double array of size $5n/2+2$, holds data needed for double-precision routines to perform TT computations. This array is initialized in the d_init_trig_transform and d_commit_trig_transform routines.
<i>spar</i>	float array of size $5n/2+2$, holds data needed for single-precision routines to perform TT computations. This array is initialized in the s_init_trig_transform and s_commit_trig_transform routines.

As *dpar* and *spar* have similar elements in respective positions, the elements are described together in [Table "Elements of the dpar and spar Arrays"](#):

Elements of the dpar and spar Arrays

Index	Description
0	Contains the first absolute tolerance used by the appropriate ?_commit_trig_transform routine. For a staggered cosine or a sine transform, $f[n]$ should be equal to 0.0 and for a staggered sine or a sine transform, $f[0]$ should be equal to 0.0. The ?_commit_trig_transform routine checks whether absolute values of these parameters are below $dpar[0]*n$ or $spar[0]*n$, depending on the routine precision. To suppress warnings resulting from tolerance checks, set $dpar[0]$ or $spar[0]$ to a sufficiently large number.
1	Reserved for future use.
2 through $5n/2+1$	Contain tabulated values of trigonometric functions. Contents of the elements depend upon the type of transform <i>tt_type</i> , set up in the ?_commit_trig_transform routine: <ul style="list-style-type: none"> If <i>tt_type</i>=<code>MKL_SINE_TRANSFORM</code>, the transform uses only the first $n/2$ array elements, which contain tabulated sine values. If <i>tt_type</i>=<code>MKL_STAGGERED_SINE_TRANSFORM</code>, the transform uses only the first $3n/2$ array elements, which contain tabulated sine and cosine values.

Index	Description
	<ul style="list-style-type: none"> • If <code>tt_type=MKL_STAGGERED2_SINE_TRANSFORM</code>, the transform uses all the $5n/2$ array elements, which contain tabulated sine and cosine values. • If <code>tt_type=MKL_COSINE_TRANSFORM</code>, the transform uses only the first n array elements, which contain tabulated cosine values. • If <code>tt_type=MKL_STAGGERED_COSINE_TRANSFORM</code>, the transform uses only the first $3n/2$ elements, which contain tabulated sine and cosine values. • If <code>tt_type=MKL_STAGGERED2_COSINE_TRANSFORM</code>, the transform uses all the $5n/2$ elements, which contain tabulated sine and cosine values.

NOTE

To save memory, you can define the array size depending upon the type of transform.

Caveat on Parameter Modifications

Flexibility of the TT interface enables you to skip a call to the `?_init_trig_transform` routine and to initialize the basic data structures explicitly in your code. You may also need to modify the contents of `ipar`, `dpar` and `spar` arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or wrong computation. You can perform a basic check for correctness and consistency of parameters by calling the `?_commit_trig_transform` routine; however, this does not ensure the correct result of a transform but only reduces the chance of errors or wrong results.

NOTE

To supply correct and consistent parameters to TT routines, you should have considerable experience in using the TT interface and good understanding of elements that the `ipar`, `spar` and `dpar` arrays contain and dependencies between values of these elements.

However, in rare occurrences, even advanced users might fail to compute a transform using TT routines after the parameter modifications. In cases like these, refer for technical support at <http://www.intel.com/software/products/support/>.

WARNING

The only way that ensures proper computation of the Trigonometric Transforms is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of `ipar`, `dpar` and `spar` arrays unless a strong need arises.

Product and Performance Information

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

Notice revision #20201201

Trigonometric Transform Implementation Details

Several aspects of the Intel® oneAPI Math Kernel Library (oneMKL) TT interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, Intel® oneAPI Math Kernel Library (oneMKL) provides you with the TT language-specific header file to include in your code:

- `mkl_trig_transforms.h`, to be used together with `mkl_dfti.h`.

NOTE

- Use of the Intel® oneAPI Math Kernel Library (oneMKL) TT software without including the above language-specific header files is not supported.
-

Header File

The header file below defines the following function prototypes:

```
void d_init_trig_transform(MKL_INT *, MKL_INT *, MKL_INT *, double *, MKL_INT *);
void d_commit_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, double *, MKL_INT *);
void d_forward_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, double *, MKL_INT *);
void d_backward_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, double *, MKL_INT *);

void s_init_trig_transform(MKL_INT *, MKL_INT *, MKL_INT *, float *, MKL_INT *);
void s_commit_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, float *, MKL_INT *);
void s_forward_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, float *, MKL_INT *);
void s_backward_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, float *, MKL_INT *);

void free_trig_transform(DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, MKL_INT *);
```

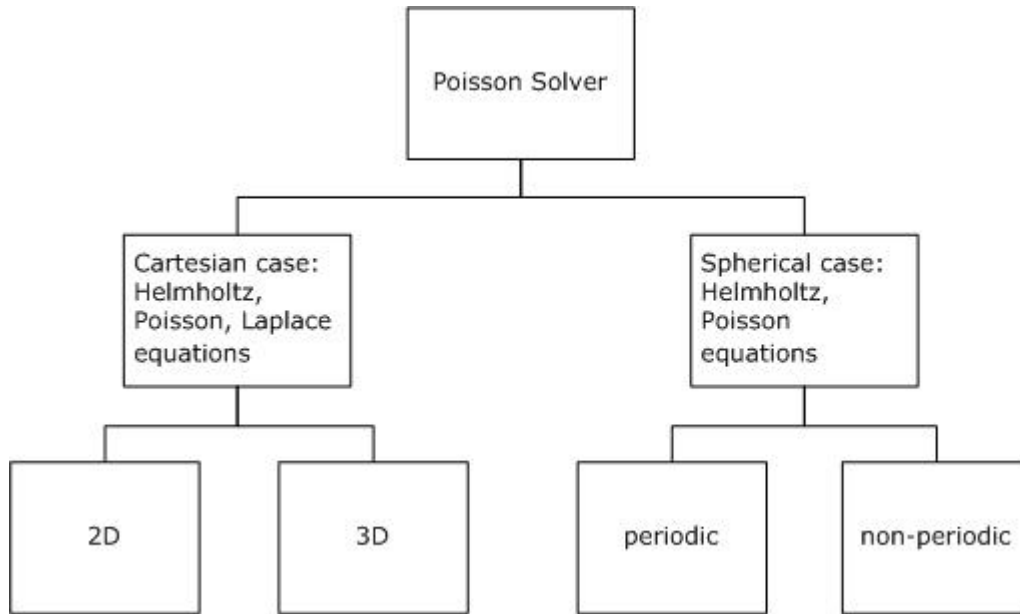
Fast Poisson Solver Routines

In addition to the Real Discrete Trigonometric Transforms (TT) interface (refer to [Trigonometric Transform Routines](#)), Intel® oneAPI Math Kernel Library (oneMKL) supports the Poisson Solver interface. This interface implements a group of routines (Poisson Solver routines) used to compute a solution of Laplace, Poisson, and Helmholtz problems of a special kind using discrete Fourier transforms. Laplace and Poisson problems are special cases of a more general Helmholtz problem. The problems that are solved by the Poisson Solver interface are defined more exactly in [Poisson Solver Implementation](#). The Poisson Solver interface provides much flexibility of use: you can call routines with the default parameter values or adjust routines to your particular needs by manually tuning routine parameters. You can adjust the style of error and warning messages to a C notation by setting up a dedicated parameter. This adds convenience to debugging, because you can read information in the way that is natural for your code. The Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver interface currently contains only routines that implement the following solvers:

- Fast Laplace, Poisson and Helmholtz solvers in a Cartesian coordinate system
- Fast Poisson and Helmholtz solvers in a spherical coordinate system.

Poisson Solver Implementation

Poisson Solver routines enable approximate solving of certain two-dimensional and three-dimensional problems. [Figure "Structure of the Poisson Solver"](#) shows the general structure of the Poisson Solver.

`__border__top`**Structure of the Poisson Solver****NOTE**

Although in the Cartesian case, both periodic and non-periodic solvers are also supported, they use the same interfaces.

Sections below provide details of the problems that can be solved using Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver.

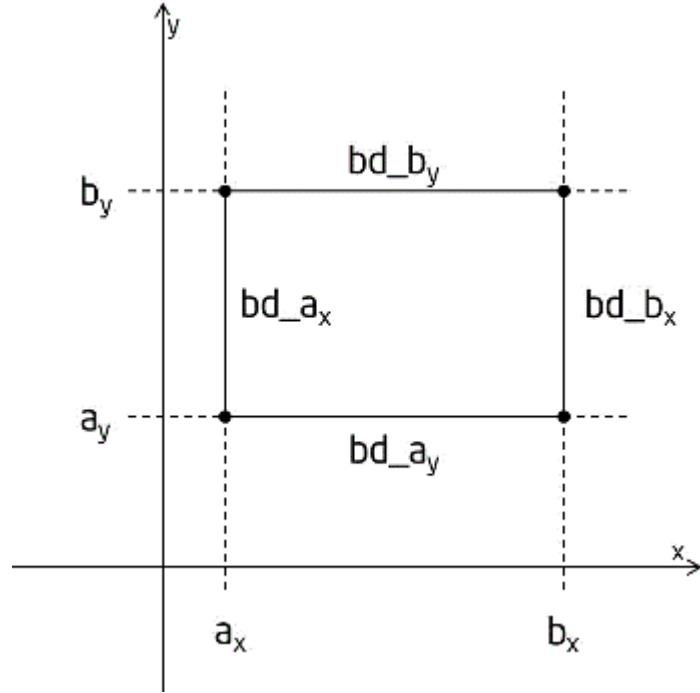
Two-Dimensional Problems**Notational Conventions**

The Poisson Solver interface description uses the following notation for boundaries of a rectangular domain $a_x < x < b_x$, $a_y < y < b_y$ on a Cartesian plane:

$$bd_a_x = \{x = a_x, a_y \leq y \leq b_y\}, \quad bd_b_x = \{x = b_x, a_y \leq y \leq b_y\}$$

$$bd_a_y = \{a_x \leq x \leq b_x, y = a_y\}, \quad bd_b_y = \{a_x \leq x \leq b_x, y = b_y\}.$$

The following figure shows these boundaries:



The wildcard "+" may stand for any of the symbols a_x , b_x , a_y , b_y , so bd_+ denotes any of the above boundaries.

The Poisson Solver interface description uses the following notation for boundaries of a rectangular domain $a_\varphi < \varphi < b_\varphi$, $a_\theta < \theta < b_\theta$ on a sphere $0 \leq \varphi \leq 2\pi$, $0 \leq \theta \leq \pi$:

$$bd_a_\varphi = \{\varphi = a_\varphi, a_\theta \leq \theta \leq b_\theta\}, \quad bd_b_\varphi = \{\varphi = b_\varphi, a_\theta \leq \theta \leq b_\theta\},$$

$$bd_a_\theta = \{a_\varphi \leq \varphi \leq b_\varphi, \theta = a_\theta\}, \quad bd_b_\theta = \{a_\varphi \leq \varphi \leq b_\varphi, \theta = b_\theta\}.$$

The wildcard "~" may stand for any of the symbols a_φ , b_φ , a_θ , b_θ , so $bd_~$ denotes any of the above boundaries.

Two-dimensional Helmholtz problem on a Cartesian plane

The two-dimensional (2D) Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + qu = f(x, y), \quad q = \text{const} \geq 0$$

in a rectangle, that is, a rectangular domain $a_x < x < b_x$, $a_y < y < b_y$, with one of the following boundary conditions on each boundary bd_+ :

- The Dirichlet boundary condition

$$u(x, y) = G(x, y)$$

- The Neumann boundary condition

$$\frac{\partial u}{\partial n}(x, y) = g(x, y)$$

where

$$n = -x \text{ on } bd_a_x, \quad n = x \text{ on } bd_b_x,$$

$$n = -y \text{ on } bd_a_y, \quad n = y \text{ on } bd_b_y.$$

- Periodic boundary conditions

$$u(a_x, y) = u(b_x, y), \quad \frac{\partial}{\partial x} u(a_x, y) = \frac{\partial}{\partial x} u(b_x, y),$$

$$u(x, a_y) = u(x, b_y), \quad \frac{\partial}{\partial y} u(x, a_y) = \frac{\partial}{\partial y} u(x, b_y).$$

Two-dimensional Poisson problem on a Cartesian plane

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The 2D Poisson problem is to find an approximate solution of the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

in a rectangle $a_x < x < b_x$, $a_y < y < b_y$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_+ . In case of a problem with the Neumann boundary condition on the entire boundary, you can find the solution of the problem only up to a constant. In this case, the Poisson Solver will compute the solution that provides the minimal Euclidean norm of a residual.

Two-dimensional (2D) Laplace problem on a Cartesian plane

The Laplace problem is a special case of the Helmholtz problem, when $q=0$ and $f(x, y)=0$. The 2D Laplace problem is to find an approximate solution of the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

in a rectangle $a_x < x < b_x$, $a_y < y < b_y$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_+ .

Helmholtz problem on a sphere

The Helmholtz problem on a sphere is to find an approximate solution of the Helmholtz equation

$$-\Delta_s u + qu = f, \quad q = \text{const} \geq 0,$$

$$\Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \phi^2} + \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial}{\partial \theta} \right)$$

in a domain bounded by angles $a_\phi \leq \phi \leq b_\phi$, $a_\theta \leq \theta \leq b_\theta$ (spherical rectangle), with boundary conditions for particular domains listed in [Table "Details of Helmholtz Problem on a Sphere"](#).

Details of Helmholtz Problem on a Sphere

Domain on a sphere	Boundary condition	Periodic/non-periodic case
Rectangular, that is, $b_\phi - a_\phi < 2\pi$ and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on each boundary bd_+	<i>non-periodic</i>
Where $a_\phi = 0$, $b_\phi = 2\pi$, and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on the boundaries bd_{a_θ} and bd_{b_θ}	<i>periodic</i>
Entire sphere, that is, $a_\phi = 0$, $b_\phi = 2\pi$, $a_\theta = 0$, and $b_\theta = \pi$	Boundary condition $\left(\sin \theta \frac{\partial u}{\partial \theta} \right) = 0$ at $\theta \rightarrow 0$ and $\theta \rightarrow \pi$ at the poles	<i>periodic</i>

Poisson problem on a sphere

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The Poisson problem on a sphere is to find an approximate solution of the Poisson equation

$$-\Delta_s u = f, \quad \Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \phi^2} + \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial}{\partial \theta} \right)$$

in a spherical rectangle $a_\varphi \leq \varphi \leq b_\varphi$, $a_\theta \leq \theta \leq b_\theta$ in cases listed in [Table "Details of Helmholtz Problem on a Sphere"](#). The solution to the Poisson problem on the entire sphere can be found up to a constant only. In this case, Poisson Solver will compute the solution that provides the minimal Euclidean norm of a residual.

Approximation of 2D problems

To find an approximate solution for any of the 2D problems, in the rectangular domain a uniform mesh can be defined for the Cartesian case as:

$$\{x_i = a_x + ih_x, y_j = a_y + jh_y\},$$

$$i = 0, \dots, n_x, j = 0, \dots, n_y, h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}$$

and for the spherical case as:

$$\{\phi_i = a_\phi + ih_\phi, \theta_j = a_\theta + jh_\theta\},$$

$$i = 0, \dots, n_\phi, j = 0, \dots, n_\theta, h_\phi = \frac{b_\phi - a_\phi}{n_\phi}, h_\theta = \frac{b_\theta - a_\theta}{n_\theta}.$$

The Poisson Solver uses the standard five-point finite difference approximation on this mesh to compute the approximation to the solution:

- In the Cartesian case, the values of the approximate solution will be computed in the mesh points (x_i, y_j) provided that you can supply the values of the right-hand side $f(x, y)$ in these points and the values of the appropriate boundary functions $G(x, y)$ and/or $g(x, y)$ in the mesh points laying on the boundary of the rectangular domain.
- In the spherical case, the values of the approximate solution will be computed in the mesh points (ϕ_i, θ_j) provided that you can supply the values of the right-hand side $f(\phi, \theta)$ in these points.

NOTE

The number of mesh intervals n_ϕ in the ϕ direction of a spherical mesh must be even in the periodic case. The Poisson Solver does not support spherical meshes that do not meet this condition.

Three-Dimensional Problems

Notational Conventions

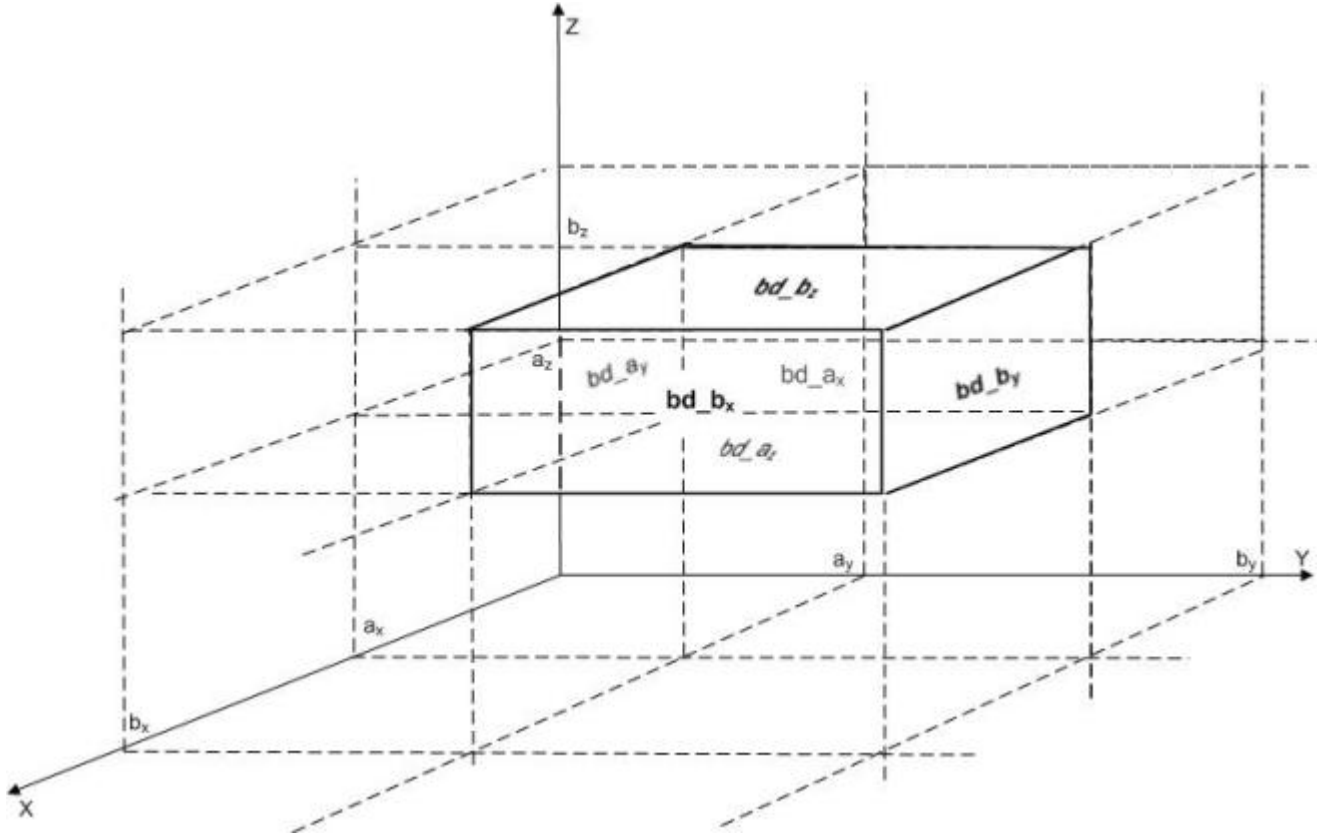
The Poisson Solver interface description uses the following notation for boundaries of a parallelepiped domain $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$:

$$bd_a_x = \{x = a_x, a_y \leq y \leq b_y, a_z \leq z \leq b_z\}, bd_b_x = \{x = b_x, a_y \leq y \leq b_y, a_z \leq z \leq b_z\},$$

$$bd_a_y = \{a_x \leq x \leq b_x, y = a_y, a_z \leq z \leq b_z\}, bd_b_y = \{a_x \leq x \leq b_x, y = b_y, a_z \leq z \leq b_z\},$$

$$bd_a_z = \{a_x \leq x \leq b_x, a_y \leq y \leq b_y, z = a_z\}, bd_b_z = \{a_x \leq x \leq b_x, a_y \leq y \leq b_y, z = b_z\}.$$

The following figure shows these boundaries:



The wildcard "+" may stand for any of the symbols $a_x, b_x, a_y, b_y, a_z, b_z$, so bd_+ denotes any of the above boundaries.

Three-dimensional (3D) Helmholtz problem

The 3D Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} + qu = f(x, y, z), \quad q = \text{const} \geq 0$$

in a parallelepiped, that is, a parallelepiped domain $a_x < x < b_x, a_y < y < b_y, a_z < z < b_z$, with one of the following boundary conditions on each boundary bd_+ :

- The Dirichlet boundary condition

$$u(x, y, z) = G(x, y, z)$$

- The Neumann boundary condition

$$\frac{\partial u}{\partial n}(x, y, z) = g(x, y, z)$$

where

$$n = -x \text{ on } bd_a_x, \quad n = x \text{ on } bd_b_x,$$

$$n = -y \text{ on } bd_a_y, \quad n = y \text{ on } bd_b_y,$$

$$n = -z \text{ on } bd_a_z, \quad n = z \text{ on } bd_b_z.$$

- Periodic boundary conditions

$$u(a_x, y, z) = u(b_x, y, z), \quad \frac{\partial}{\partial x} u(a_x, y, z) = \frac{\partial}{\partial x} u(b_x, y, z),$$

$$u(x, a_y, z) = u(x, b_y, z), \quad \frac{\partial}{\partial y} u(x, a_y, z) = \frac{\partial}{\partial y} u(x, b_y, z),$$

$$u(x, y, a_z) = u(x, y, b_z), \frac{\partial}{\partial z} u(x, y, a_z) = \frac{\partial}{\partial z} u(x, y, b_z).$$

Three-dimensional (3D) Poisson problem

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The 3D Poisson problem is to find an approximate solution of the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f(x, y, z)$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_{-+} .

Three-dimensional (3D) Laplace problem

The Laplace problem is a special case of the Helmholtz problem, when $q=0$ and $f(x, y, z)=0$. The 3D Laplace problem is to find an approximate solution of the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_{-+} .

Approximation of 3D problems

To find an approximate solution for each of the 3D problems, a uniform mesh can be defined in the parallelepiped domain as:

$$\{x_i = a_x + ih_x, y_j = a_y + jh_y, z_k = a_z + kh_z\},$$

where

$$i = 0, \dots, n_x, j = 0, \dots, n_y, k = 0, \dots, n_z,$$

$$h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}, h_z = \frac{b_z - a_z}{n_z}.$$

The Poisson Solver uses the standard seven-point finite difference approximation on this mesh to compute the approximation to the solution. The values of the approximate solution will be computed in the mesh points (x_i, y_j, z_k) , provided that you can supply the values of the right-hand side $f(x, y, z)$ in these points and the values of the appropriate boundary functions $G(x, y, z)$ and/or $g(x, y, z)$ in the mesh points laying on the boundary of the parallelepiped domain.

Sequence of Invoking Poisson Solver Routines

NOTE

This description always shows the solution process for the Helmholtz problem, because Fast Poisson Solvers and Fast Laplace Solvers are special cases of Fast Helmholtz Solvers (see [Poisson Solver Implementation](#)).

The Poisson Solver interface enables you to compute a solution of the Helmholtz problem in four steps. Each step is performed by a dedicated routine. [Table "Poisson Solver Interface Routines"](#) lists the routines and briefly describes their purpose.

Most Poisson Solver routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names. The routines for the Cartesian coordinate system have 2D and 3D versions. Their names end respectively in "2D" and "3D". The routines for spherical coordinate system have periodic and non-periodic versions. Their names end respectively in "p" and "np".

Poisson Solver Interface Routines

Routine	Description
<code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/ ?_init_sph_p/?_init_sph_np</code>	Initializes basic data structures for Fast Helmholtz Solver in the 2D/3D/periodic/non-periodic case, respectively.
<code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/ ?_commit_sph_p/?_commit_sph_np</code>	Checks consistency and correctness of input data and initializes data structures for the solver, including those used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface ¹ .
<code>?_Helmholtz_2D/?_Helmholtz_3D/?_sph_p/?_sph_np</code>	Computes an approximate solution of the 2D/3D/periodic/non-periodic Helmholtz problem (see Poisson Solver Implementation) specified by the parameters.
<code>free_Helmholtz_2D/free_Helmholtz_3D/ free_sph_p/free_sph_np</code>	Releases the memory used by the data structures needed for calling the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface ¹ .

¹Poisson Solver routines call the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface for better performance.

To find an approximate solution of Helmholtz problem only once, the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver interface routines are normally invoked in the order in which they are listed in [Table "Poisson Solver Interface Routines"](#).

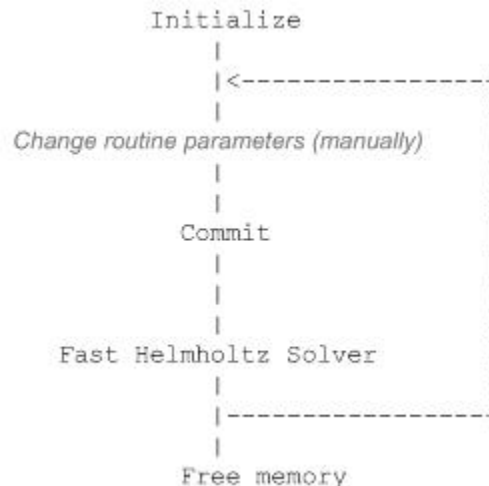
NOTE

Though the order of invoking Poisson Solver routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in [Figure "Typical Order of Invoking Poisson Solver Routines"](#) indicates the typical order in which Poisson Solver routines can be invoked in a general case.

__border__ top

Typical Order of Invoking Poisson Solver Routines



A general scheme of using Poisson Solver routines for double-precision computations in a 3D Cartesian case is shown below. You can change this scheme to a scheme for single-precision computations by changing the initial letter of the Poisson Solver routine names from "d" to "s". You can also change the scheme below from the 3D to 2D case by changing the ending of the Poisson Solver routine names.

```
...
d_init_Helmholtz_3D(&ax, &bx, &ay, &by, &az, &bz, &nx, &ny, &nz, BCtype, ipar, dpar, &stat);
/* change parameters in ipar and/or dpar if necessary. */
/* note that the result of the Fast Helmholtz Solver will be in f. If you want to keep the data
that is stored in f, save it to another location before the function call below */
d_commit_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar,
dpar, &stat);
d_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar, dpar,
&stat);
free_Helmholtz_3D (&xhandle, &yhandle, ipar, &stat);
/* here you may clean the memory used by f, dpar, ipar */
...
```

A general scheme of using Poisson Solver routines for double-precision computations in a spherical periodic case is shown below. You can change this scheme to a scheme for single-precision computations by changing the initial letter of the Poisson Solver routine names from "d" to "s". You can also change the scheme below to a scheme for a non-periodic case by changing the ending of the Poisson Solver routine names from "p" to "np".

```
...
d_init_sph_p(&ap, &bp, &at, &bt, &np, &nt, &q, ipar, dpar, &stat);
/* change parameters in ipar and/or dpar if necessary. */
/* note that the result of the Fast Helmholtz Solver will be in f. If you want to keep the data
that is stored in f, save it to another location before the function call below */
d_commit_sph_p(f, &handle_s, &handle_c, ipar, dpar, &stat);
d_sph_p(f, &handle_s, &handle_c, ipar, dpar, &stat);
free_sph_p(&handle_s, &handle_c, ipar, &stat);
/* here you may clean the memory used by f, dpar, ipar */
...
```

You can find examples of code that uses Poisson Solver routines to solve Helmholtz problem (in both Cartesian and spherical cases) in the `examples\pdepoissonc\source` folder in your Intel® oneAPI Math Kernel Library (oneMKL) directory.

Fast Poisson Solver Interface Description

All numerical types in this section are either standard C types `float` and `double` or `MKL_INT` integer type. For more information on the C types, refer to [C Datatypes Specific to oneMKL](#) and the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*. To better understand usage of the types, see examples in the `examples\pdepoissonc\source` folder in your Intel® oneAPI Math Kernel Library (oneMKL) directory.

Routine Options

All Poisson Solver routines use parameters for passing various options to the routines. These parameters are arrays `ipar`, `dpar`, and `spar`. Values for these parameters should be specified very carefully (see [Common Parameters](#)). You can change these values during computations to meet your needs. For more details, see the descriptions of specific routines.

WARNING

To avoid failure or incorrect results, you must provide correct and consistent parameters to the routines.

	float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
	The coordinate of the leftmost boundary of the domain along the x-axis.
<i>bx</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
	The coordinate of the rightmost boundary of the domain along the x-axis.
<i>ay</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
	The coordinate of the leftmost boundary of the domain along the y-axis.
<i>by</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
	The coordinate of the rightmost boundary of the domain along the y-axis.
<i>az</i>	double* for d_init_Helmholtz_3D, float* for s_init_Helmholtz_3D.
	The coordinate of the leftmost boundary of the domain along the z-axis. This parameter is needed only for the ?_init_Helmholtz_3D routine.
<i>bz</i>	double* for d_init_Helmholtz_3D, float* for s_init_Helmholtz_3D.
	The coordinate of the rightmost boundary of the domain along the z-axis. This parameter is needed only for the ?_init_Helmholtz_3D routine.
<i>nx</i>	MKL_INT*. The number of mesh intervals along the x-axis.
<i>ny</i>	MKL_INT*. The number of mesh intervals along the y-axis.
<i>nz</i>	MKL_INT*. The number of mesh intervals along the z-axis. This parameter is needed only for the ?_init_Helmholtz_3D routine.
<i>BCtype</i>	char*. Contains the type of boundary conditions on each boundary. Must contain four characters for ?_init_Helmholtz_2D and six characters for ?_init_Helmholtz_3D. Each of the characters can be 'N' (Neumann boundary condition), 'D' (Dirichlet boundary condition), or 'P' (periodic boundary conditions). Specify the types of boundary conditions for the boundaries in the following order: <i>bd_ax</i> , <i>bd_bx</i> , <i>bd_ay</i> , <i>bd_by</i> , <i>bd_az</i> , and <i>bd_bz</i> . Specify periodic boundary conditions on the respective boundaries in pairs (for example, 'PPDD' or 'NNPP' in the 2D case). The types of boundary conditions for the last two boundaries are needed only in the 3D case.
<i>q</i>	double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.

The constant Helmholtz coefficient. Note that to solve Poisson or Laplace problem, you should set the value of q to 0.

Output Parameters

<i>ipar</i>	MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to ipar).
<i>dpar</i>	double array of size $5 \times nx/2 + 7$ in the 2D case or $5 \times (nx + ny)/2 + 9$ in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).
<i>spar</i>	float array of size $5 \times nx/2 + 7$ in the 2D case or $5 \times (nx + ny)/2 + 9$ in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> . Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routines initialize basic data structures for Poisson Solver computations of the appropriate precision. All routines invoked after a call to a `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine use values of the *ipar*, *dpar* and *spar* array parameters returned by the routine. Detailed description of the array parameters can be found in [Common Parameters](#).

Caution

Data structures initialized and created by 2D flavors of the routine cannot be used by 3D flavors of any Poisson Solver routines, and vice versa.

You can skip calls to these routines in your code. However, see [Caveat on Parameter Modifications](#) for information on initializing the data structures.

Return Values

<i>stat</i> = 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this <i>stat</i> value.
<i>stat</i> = -99999	The routine failed to complete the task because of a fatal error.

`_commit_Helmholtz_2D/?_commit_Helmholtz_3D`

Checks consistency and correctness of input data and initializes certain data structures required to solve 2D/3D Helmholtz problem.

Syntax

```
void d_commit_Helmholtz_2D (double * f, const double * bd_ax, const double * bd_bx,
const double * bd_ay, const double * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT *
ipar, double * dpar, MKL_INT * stat );
```

```

void s_commit_Helmholtz_2D (float * f, const float * bd_ax, const float * bd_bx, const
float * bd_ay, const float * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar,
float * spar, MKL_INT * stat );

void d_commit_Helmholtz_3D (double * f, const double * bd_ax, const double * bd_bx,
const double * bd_ay, const double * bd_by, const double * bd_az, const double * bd_bz,
DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar,
double * dpar, MKL_INT * stat );

void s_commit_Helmholtz_3D (float * f, const float * bd_ax, const float * bd_bx, const
float * bd_ay, const float * bd_by, const float * bd_az, const float * bd_bz,
DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar,
float * spar, MKL_INT * stat );

```

Include Files

- mkl.h

Input Parameters

<i>f</i>	<p>double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.</p> <p>Contains the right-hand side of the problem packed in a single vector:</p> <ul style="list-style-type: none"> • 2D problem: The size of the vector for the is $(n_x+1)*(n_y+1)$. The value of the right-hand side in the mesh point (i, j) is stored in $f[i+j*(n_x+1)]$. • 3D problem: The size of the vector for the is $(n_x+1)*(n_y+1)*(n_z+1)$. The value of the right-hand side in the mesh point (i, j, k) is stored in $f[i+j*(n_x+1)+k*(n_x+1)*(n_y+1)]$. <p>Note that to solve the Laplace problem, you should set all the elements of the array <i>f</i> to 0.</p> <p>Note also that the array <i>f</i> may be altered by the routine. To preserve the <i>f</i> vector, save it to another memory location.</p>
<i>ipar</i>	<p>MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver (for details, refer to ipar).</p>
<i>dpar</i>	<p>double array of size depending on the dimension of the problem:</p> <ul style="list-style-type: none"> • 2D problem: $5*n_x/2+7$ • 3D problem: $5*(n_x+n_y)/2+9$ <p>Contains double-precision data to be used by the Fast Helmholtz Solver (for details, refer to dpar and spar).</p>
<i>spar</i>	<p>float array of size depending on the dimension of the problem:</p> <ul style="list-style-type: none"> • 2D problem: $5*n_x/2+7$ • 3D problem: $5*(n_x+n_y)/2+9$ <p>Contains single-precision data to be used by the Fast Helmholtz Solver (for details, refer to dpar and spar).</p>
<i>bd_ax</i>	<p>double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.</p>

Contains values of the boundary condition on the leftmost boundary of the domain along the x-axis (for more information, refer to [a detailed description of *bd_ax*](#)).

bd_bx

double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D,
float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.

Contains values of the boundary condition on the rightmost boundary of the domain along the x-axis (for more information, refer to [a detailed description of *bd_bx*](#)).

bd_ay

double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D,
float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.

Contains values of the boundary condition on the leftmost boundary of the domain along the y-axis (for more information, refer to [a detailed description of *bd_ay*](#)).

bd_by

double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D,
float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.

Contains values of the boundary condition on the rightmost boundary of the domain along the y-axis (for more information, refer to [a detailed description of *bd_by*](#)).

bd_az

double* for d_commit_Helmholtz_3D,
float* for s_commit_Helmholtz_3D.

Used only by ?_commit_Helmholtz_3D. Contains values of the boundary condition on the leftmost boundary of the domain along the z-axis (for more information, refer to [a detailed description of *bd_az*](#)).

bd_bz

double* for d_commit_Helmholtz_3D,
float* for s_commit_Helmholtz_3D.

Used only by ?_commit_Helmholtz_3D. Contains values of the boundary condition on the rightmost boundary of the domain along the z-axis (for more information, refer to [a detailed description of *bd_bz*](#)).

Output Parameters

f

Contains right-hand side of the problem, possibly altered on output.

ipar

Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in [ipar](#).

dpar

Contains double-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in [dpar](#) and [spar](#).

spar

Contains single-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in [dpar](#) and [spar](#).

xhandle, yhandle

DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer to [FFT Functions](#)). *yhandle* is used only by ?_commit_Helmholtz_3D.

stat MKL_INT*. Routine completion status, which is also written to *ipar[0]*. Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routines check the consistency and correctness of the parameters to be passed to the solver routines `?_Helmholtz_2D/?_Helmholtz_3D`. They also initialize the *xhandle* and *yhandle* data structures, *ipar* array, and *dpar* or *spar* array, depending upon the routine precision. Refer to [Common Parameters](#) to find out which particular array elements the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routines initialize and to what values these elements are initialized.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of Poisson Solver routines, see [Caveat on Parameter Modifications](#).

Unlike `?_init_Helmholtz_2D/?_init_Helmholtz_3D`, you must call

the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routines in your code. Values of *ax*, *bx*, *ay*, *by*, *az*, and *bz* are passed to the routines with the *spar/dpar* array, and values of *nx*, *ny*, *nz*, and *BCtype* are passed with the *ipar* array.

Return Values

<i>stat</i> = 1	The routine completed without errors but with warnings.
<i>stat</i> = 0	The routine successfully completed the task.
<i>stat</i> = -100	The routine stopped because an error in the input data was found, or the data in the <i>dpar</i> , <i>spar</i> , or <i>ipar</i> array was altered by mistake.
<i>stat</i> = -1000	The routine stopped because of an Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
<i>stat</i> = -10000	The routine stopped because the initialization failed to complete or the parameter <i>ipar[0]</i> was altered by mistake.
<i>stat</i> = -99999	The routine failed to complete the task because of a fatal error.

`?_Helmholtz_2D/?_Helmholtz_3D`

Computes the solution of the 2D/3D Helmholtz problem specified by the parameters.

Syntax

```
void d_Helmholtz_2D (double * f, const double * bd_ax, const double * bd_bx, const
double * bd_ay, const double * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar,
const double * dpar, MKL_INT * stat );
```

```
void s_Helmholtz_2D (float * f, const float * bd_ax, const float * bd_bx, const float *
bd_ay, const float * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar, const
float * spar, MKL_INT * stat );
```

```
void d_Helmholtz_3D (double * f, const double * bd_ax, const double * bd_bx, const
double * bd_ay, const double * bd_by, const double * bd_az, const double * bd_bz,
DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar,
const double * dpar, MKL_INT * stat );
```

```
void s_Helmholtz_3D (float * f, const float * bd_ax, const float * bd_bx, const float *
bd_ay, const float * bd_by, const float * bd_az, const float * bd_bz,
DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar,
const float * spar, MKL_INT * stat );
```

Include Files

- mkl.h

Input Parameters

<i>f</i>	<p>double* for d_Helmholtz_2D/d_Helmholtz_3D, float* for s_Helmholtz_2D/s_Helmholtz_3D.</p> <p>Contains the right-hand side of the problem packed in a single vector and modified by the appropriate ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine. Note that an attempt to substitute the original right-hand side vector, which was passed to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine, at this point results in an incorrect solution.</p> <ul style="list-style-type: none"> • 2D problem: the size of the vector is $(nx+1)*(ny+1)$. The value of the modified right-hand side in the mesh point (i, j) is stored in $f[i+j*(nx+1)]$. • 3D problem: the size of the vector is $(nx+1)*(ny+1)*(nz+1)$. The value of the modified right-hand side in the mesh point (i, j, k) is stored in $f[i+j*(nx+1)+k*(nx+1)*(ny+1)]$.
<i>xhandle, yhandle</i>	<p>DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer to FFT Functions). <i>yhandle</i> is used only by ?_Helmholtz_3D.</p>
<i>ipar</i>	<p>MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to ipar).</p>
<i>dpar</i>	<p>double array of size depending on the dimension of the problem:</p> <ul style="list-style-type: none"> • 2D problem: $5*nx/2+7$ • 3D problem: $5*(nx+ny)/2+9$ <p>Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).</p>
<i>spar</i>	<p>float array of size depending on the dimension of the problem:</p> <ul style="list-style-type: none"> • 2D problem: $5*nx/2+7$ • 3D problem: $5*(nx+ny)/2+9$ <p>Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).</p>
<i>bd_ax</i>	<p>double* for d_Helmholtz_2D/d_Helmholtz_3D, float* for s_Helmholtz_2D/s_Helmholtz_3D.</p> <p>Contains values of the boundary condition on the leftmost boundary of the domain along the x-axis (for more information, refer to a detailed description of bd_ax).</p>

<i>bd_bx</i>	double* for d_Helmholtz_2D/d_Helmholtz_3D, float* for s_Helmholtz_2D/s_Helmholtz_3D. <p>Contains values of the boundary condition on the rightmost boundary of the domain along the x-axis (for more information, refer to a detailed description of <i>bd_bx</i>).</p>
<i>bd_ay</i>	double* for d_Helmholtz_2D/d_Helmholtz_3D, float* for s_Helmholtz_2D/s_Helmholtz_3D. <p>Contains values of the boundary condition on the leftmost boundary of the domain along the y-axis for more information, refer to a detailed description of <i>bd_ay</i>).</p>
<i>bd_by</i>	double* for d_Helmholtz_2D/d_Helmholtz_3D, float* for s_Helmholtz_2D/s_Helmholtz_3D. <p>Contains values of the boundary condition on the rightmost boundary of the domain along the y-axis (for more information, refer to a detailed description of <i>bd_by</i>).</p>
<i>bd_az</i>	double* for d_Helmholtz_3D, float* for s_Helmholtz_3D. <p>Used only by ?_Helmholtz_3D. Contains values of the boundary condition on the leftmost boundary of the domain along the z-axis (for more information, refer to a detailed description of <i>bd_az</i>).</p>
<i>bd_bz</i>	double* for d_Helmholtz_3D, float* for s_Helmholtz_3D. <p>Used only by ?_Helmholtz_3D. Contains values of the boundary condition on the rightmost boundary of the domain along the z-axis (for more information, refer to a detailed description of <i>bd_bz</i>).</p>

NOTE

To avoid incorrect computation results, do not change arrays *bd_ax*, *bd_bx*, *bd_ay*, *bd_by*, *bd_az*, *bd_bz* between a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine and a subsequent call to the appropriate ?_Helmholtz_2D/?_Helmholtz_3D routine.

Output Parameters

<i>f</i>	On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.
<i>xhandle</i> , <i>yhandle</i>	Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface. Although the addresses do not change, the structures are modified on output.
<i>ipar</i>	Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in ipar .

`stat` MKL_INT*. Routine completion status, which is also written to `ipar[0]`. Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_Helmholtz_2D/?_Helmholtz_3D` routines compute the approximate solution of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to formulas given in [Poisson Solver Implementation](#). The `f` parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of `ax`, `bx`, `ay`, `by`, `az`, and `bz` are passed to the routines with the `spar/dpar` array, and values of `nx`, `ny`, `nz`, and `BCtype` are passed with the `ipar` array.

Return Values

<code>stat= 1</code>	The routine completed without errors but with some warnings.
<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -2</code>	The routine stopped because division by zero occurred. It usually happens if the data in the <code>dpar</code> or <code>spar</code> array was altered by mistake.
<code>stat= -3</code>	The routine stopped because the sufficient memory was unavailable for the computations.
<code>stat= -100</code>	The routine stopped because an error in the input data was found or the data in the <code>dpar</code> , <code>spar</code> , or <code>ipar</code> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of the Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

free_Helmholtz_2D/free_Helmholtz_3D

Releases the memory allocated for the data structures used by the FFT interface.

Syntax

```
void free_Helmholtz_2D(DFTI_DESCRIPTOR_HANDLE* xhandle, MKL_INT* ipar, MKL_INT* stat);
void free_Helmholtz_3D(DFTI_DESCRIPTOR_HANDLE* xhandle, DFTI_DESCRIPTOR_HANDLE*
yhandle, MKL_INT* ipar, MKL_INT* stat);
```

Include Files

- `mkh.h`

Input Parameters

<i>xhandle, yhandle</i>	DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer to FFT Functions). The structure <i>yhandle</i> is used only by <code>free_Helmholtz_3D</code> .
<i>ipar</i>	MKL_INT array of size 128. Contains integer data used by Fast Helmholtz Solver (for details, refer to ipar).

Output Parameters

<i>xhandle, yhandle</i>	Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface. Memory allocated for the structures is released on output.
<i>ipar</i>	Contains integer data used by Fast Helmholtz Solver. On output, the status of the routine call is written to <i>ipar[0]</i> .
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> .

Description

The `free_Helmholtz_2D-free_Helmholtz_3D` routine releases the memory used by the *xhandle* and *yhandle* structures, which are needed for calling the Intel® oneAPI Math Kernel Library (oneMKL) FFT functions. To release memory allocated for other parameters, include memory release statements in your code.

Return Values

<i>stat</i> = 0	The routine successfully completed the task.
<i>stat</i> = -1000	The routine stopped because of an Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
<i>stat</i> = -99999	The routine failed to complete the task because of a fatal error.

Routines for the Spherical Solver

The section describes Poisson Solver routines for the spherical case, their syntax, parameters, and return values. All flavors of the same routine are described together: single- and double-precision and periodic (having names ending in "p") and non-periodic (having names ending in "np").

These Poisson Solver routines also call the Intel® oneAPI Math Kernel Library (oneMKL) FFT routines (described in [FFT Functions](#)), which enhance the performance of the Poisson Solver routines.

[?_init_sph_p/?_init_sph_np](#)

Initializes basic data structures of the periodic and non-periodic Fast Helmholtz Solver on a sphere.

Syntax

```
void d_init_sph_p (const double * ap, const double * at, const double * bp, const
double * bt, const MKL_INT * np, const MKL_INT * nt, const double * q, MKL_INT * ipar,
double * dpar, MKL_INT * stat );
```

```
void s_init_sph_p (const float * ap, const float * at, const float * bp, const float *
bt, const MKL_INT * np, const MKL_INT * nt, const float * q, MKL_INT * ipar, float *
spar, MKL_INT * stat );
```

```
void d_init_sph_np (const double * ap, const double * at, const double * bp, const
double * bt, const MKL_INT * np, const MKL_INT * nt, const double * q, MKL_INT * ipar,
double * dpar, MKL_INT * stat );
```

```
void s_init_sph_np (const float * ap, const float * at, const float * bp, const float *
bt, const MKL_INT * np, const MKL_INT * nt, const float * q, MKL_INT * ipar, float *
spar, MKL_INT * stat );
```

Include Files

- mkl.h

Input Parameters

<i>ap</i>	double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np. The coordinate (angle) of the leftmost boundary of the domain along the ϕ -axis.
<i>bp</i>	double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np. The coordinate (angle) of the rightmost boundary of the domain along the ϕ -axis.
<i>at</i>	double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np. The coordinate (angle) of the leftmost boundary of the domain along the θ -axis.
<i>bt</i>	double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np. The coordinate (angle) of the rightmost boundary of the domain along the θ -axis.
<i>np</i>	MKL_INT*. The number of mesh intervals along the ϕ -axis. Must be even in the periodic case.
<i>nt</i>	MKL_INT*. The number of mesh intervals along the θ -axis.
<i>q</i>	double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np. The constant Helmholtz coefficient. To solve the Poisson problem, set the value of <i>q</i> to 0.

Output Parameters

<i>ipar</i>	MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to ipar).
-------------	---

<i>dpar</i>	double array of size $5*np/2+nt+10$. Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
<i>spar</i>	float array of size $5*np/2+nt+10$. Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> . Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_init_sph_p/?_init_sph_np` routines initialize basic data structures for Poisson Solver computations. All routines invoked after a call to a `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine use values of the *ipar*, *dpar*, and *spar* array parameters returned by the routine. A detailed description of the array parameters can be found in [Common Parameters](#).

Caution

Data structures initialized and created by periodic flavors of the routine cannot be used by non-periodic flavors of any Poisson Solver routines for Helmholtz Solver on a sphere, and vice versa.

You can skip calls to these routines in your code. However, see [Caveat on Parameter Modifications](#) for information on initializing the data structures.

Return Values

<i>stat</i> = 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this <i>stat</i> value.
<i>stat</i> = -99999	The routine failed to complete the task because of fatal error.

[?_commit_sph_p/?_commit_sph_np](#)

Checks consistency and correctness of input data and initializes certain data structures required to solve the periodic/non-periodic Helmholtz problem on a sphere.

Syntax

```
void d_commit_sph_p(double* f, DFTI_DESCRIPTOR_HANDLE* handle_s,
DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, double* dpar, MKL_INT* stat);

void s_commit_sph_p(float* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE*
handle_c, MKL_INT* ipar, float* spar, MKL_INT* stat);

void d_commit_sph_np(double* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, double*
dpar, MKL_INT* stat);

void s_commit_sph_np(float* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, float*
spar, MKL_INT* stat);
```


Include Files

- `mkl.h`

Input Parameters

<i>f</i>	double* for <code>d_commit_sph_p/d_commit_sph_np</code> , float* for <code>s_commit_sph_p/s_commit_sph_np</code> . Contains the right-hand side of the problem packed in a single vector. The size of the vector is $(np+1)*(nt+1)$ and value of the right-hand side in the mesh point (i, j) is stored in $f[i+j*(np+1)]$. Note that the array <i>f</i> may be altered by the routine. Save this vector to another memory location if you want to preserve it.
<i>ipar</i>	MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to ipar).
<i>dpar</i>	double array of size $5*np/2+nt+10$. Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
<i>spar</i>	float array of size $5*np/2+nt+10$. Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).

Output Parameters

<i>f</i>	Contains the right-hand side of the problem, possibly altered on output.
<i>ipar</i>	Contains integer data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in ipar .
<i>dpar</i>	Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar .
<i>spar</i>	Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar .
<i>handle_s, handle_c, handle</i>	DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer to FFT Functions). <i>handle_s</i> and <i>handle_c</i> are used only in <code>?_commit_sph_p</code> and <i>handle</i> is used only in <code>?_commit_sph_np</code> .
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> . Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_commit_sph_p/?_commit_sph_np` routines check consistency and correctness of the parameters to be passed to the solver routines `?_sph_p/?_sph_np`, respectively. They also initialize certain data structures. The routine `?_commit_sph_p` initializes structures *handle_s* and *handle_c*, and `?_commit_sph_np` initializes *handle*. The routines also initialize the *ipar* array and *dpar* or *spar* array,

depending upon the routine precision. Refer to [Common Parameters](#) to find out which particular array elements the `?_commit_sph_p`/`?_commit_sph_np` routines initialize and to what values these elements are initialized.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of Poisson Solver routines, see [Caveat on Parameter Modifications](#).

Unlike `?_init_sph_p/?_init_sph_np`, you must call the `?_commit_sph_p/?_commit_sph_np` routines. Values of `np` and `nt` are passed to each of the routines with the `ipar` array.

Return Values

<code>stat= 1</code>	The routine completed without errors but with warnings.
<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -100</code>	The routine stopped because an error in the input data was found or the data in the <i>dpar</i> , <i>spar</i> , or <i>ipar</i> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of an Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <i>ipar</i> [0] was altered by mistake.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

?_sph_p/?_sph_np

Computes the solution of the spherical Helmholtz problem specified by the parameters.

Syntax

```
void d_sph_p(double* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE*
handle_c, MKL_INT* ipar, double* dpar, MKL_INT* stat);

void s_sph_p(float* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE*
handle_c, MKL_INT* ipar, float* spar, MKL_INT* stat);

void d_sph_np(double* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, double* dpar,
MKL_INT* stat);

void s_sph_np(float* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, float* spar,
MKL_INT* stat);
```

Include Files

- `mk1.h`

Input Parameters

```
f                                double* for d_sph_p/d_sph_np,  
                                float* for s sph p/s sph np.
```

Contains the right-hand side of the problem packed in a single vector and modified by the appropriate `?_commit_sph_p/?_commit_sph_np` routine. Note that an attempt to substitute the original right-hand side vector, which was passed to the `?_commit_sph_p/?_commit_sph_np` routine, at this point results in an incorrect solution.

The size of the vector is $(np+1)*(nt+1)$ and the value of the modified right-hand side in the mesh point (i, j) is stored in $f[i+j*(np+1)]$.

handle_s, handle_c, handle

DFTI_DESCRIPTOR_HANDLE*. Data structures used by Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer to [FFT Functions](#)). *handle_s* and *handle_c* are used only in `?_sph_p` and *handle* is used only in `?_sph_np`.

ipar

MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to [ipar](#)).

dpar

double array of size $5*np/2+nt+10$. Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to [dpar](#) and [spar](#)).

spar

float array of size $5*np/2+nt+10$. Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to [dpar](#) and [spar](#)).

Output Parameters

f

On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.

handle_s, handle_c, handle

Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface.

ipar

Contains integer data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in [ipar](#).

dpar

Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in [dpar](#) and [spar](#).

spar

Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in [dpar](#) and [spar](#).

stat

MKL_INT*. Routine completion status, which is also written to *ipar[0]*. Continue to call other Poisson Solver routines only if the status is 0.

Description

The `sph_p/sph_np` routines compute the approximate solution on a sphere of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to the formulas given in [Poisson Solver Implementation](#). The *f* parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of *np* and *nt* are passed to each of the routines with the *ipar* array.

Return Values

<code>stat= 1</code>	The routine completed without errors but with warnings.
<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -2</code>	The routine stopped because division by zero occurred. It usually happens if the data in the <code>dpar</code> or <code>spar</code> array was altered by mistake.
<code>stat= -3</code>	The routine stopped because the memory was insufficient to complete the computations.
<code>stat= -100</code>	The routine stopped because an error in the input data was found or the data in the <code>dpar</code> , <code>spar</code> , or <code>ipar</code> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of an Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

free_sph_p/free_sph_np

Releases the memory allocated for the data structures used by the FFT interface.

Syntax

```
void free_sph_p(DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c,
MKL_INT* ipar, MKL_INT* stat);
```

```
void free_sph_np(DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, MKL_INT* stat);
```

Include Files

- `mkh.h`

Input Parameters

<code>handle_s</code> , <code>handle_c</code> , <code>handle</code>	DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface (for details, refer to FFT Functions). The structures <code>handle_s</code> and <code>handle_c</code> are used only in <code>free_sph_p</code> , and <code>handle</code> is used only in <code>free_sph_np</code> .
<code>ipar</code>	MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to ipar).

Output Parameters

<code>handle_s</code> , <code>handle_c</code> , <code>handle</code>	Data structures used by the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface. Memory allocated for the structures is released on output.
---	--

<i>ipar</i>	Contains integer data to be used by Fast Helmholtz Solver on a sphere. On output, the status of the routine call is written to <i>ipar[0]</i> .
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> .

Description

The `free_sph_p/free_sph_np` routine releases the memory used by the `handle_s`, `handle_c` or `handle` structures, needed for calling the Intel® oneAPI Math Kernel Library (oneMKL) FFT functions. To release memory allocated for other parameters, include memory release statements in your code.

Return Values

<i>stat</i> = 0	The routine successfully completed the task.
<i>stat</i> = -1000	The routine stopped because of an Intel® oneAPI Math Kernel Library (oneMKL) FFT or TT interface error.
<i>stat</i> = -99999	The routine failed to complete the task because of a fatal error.

Common Parameters for the Poisson Solver

ipar

<i>ipar</i>	MKL_INT array of size 128, holds integer data needed for Fast Helmholtz Solver (both for Cartesian and spherical coordinate systems). Its elements are described in Table "Elements of the ipar Array" :
-------------	--

NOTE

Initial values can be assigned to the array parameters by the appropriate `?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np` and `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np` routines.

Elements of the ipar Array

Index	Description
0	<p>Contains status value of the last Poisson Solver routine called. In general, it should be 0 on exit from a routine to proceed with the Fast Helmholtz Solver. The element has no predefined values. This element can also be used to inform the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np</code> routines of how the Commit step of the computation should be carried out (see Figure "Typical Order of Invoking Poisson Solver Routines"). A non-zero value of <i>ipar[0]</i> with decimal representation</p> $\overline{abc} = 100a + 10b + c$ <p>=100<i>a</i>+10<i>b</i>+<i>c</i>, where each of <i>a</i>, <i>b</i>, and <i>c</i> is equal to 0 or 9, indicates that some parts of the Commit step should be omitted.</p> <ul style="list-style-type: none"> • If <i>c</i>=9, the routine omits checking of parameters and initialization of the data structures. • If <i>b</i>=9,

Index	Description
	<ul style="list-style-type: none"> In the Cartesian case, the routine omits the adjustment of the right-hand side vector \vec{f} to the Neumann boundary condition (multiplication of boundary values by 0.5 as well as incorporation of the boundary function g) and/or the Dirichlet boundary condition (setting boundary values to 0 as well as incorporation of the boundary function G). For the Helmholtz solver on a sphere, the routine omits computation of the spherical weights for the $dpar/spar$ array. If $a=9$, the routine omits the normalization of the right-hand side vector \vec{f}. Depending on the solver, the normalization means: <ul style="list-style-type: none"> 2D Cartesian case: multiplication by h_y^2, where h_y is the mesh size in the y direction (for details, see Poisson Solver Implementation). 3D (Cartesian) case: multiplication by h_z^2, where h_z is the mesh size in the z direction. Helmholtz solver on a sphere: multiplication by h_θ^2, where h_θ is the mesh size in the θ direction (for details, see Poisson Solver Implementation). <p>Using <code>ipar[0]</code> you can adjust the routine to your needs and improve efficiency in solving multiple Helmholtz problems that differ only in the right-hand side. You must be cautious when using this method, because any misunderstanding of the commit process may cause incorrect results or program failure (see also Caveat on Parameter Modifications).</p>
1	<p>Contains error messaging options:</p> <ul style="list-style-type: none"> <code>ipar[1]=-1</code> indicates that all error messages are printed to the <code>MKL_Poisson_Library_log.txt</code> file in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device (usually, screen). <code>ipar[1]=0</code> indicates that no error messages will be printed. <code>ipar[1]=1</code> is the default value. It indicates that all error messages are printed to the standard output device. <p>In case of errors, the <code>stat</code> parameter contains a non-zero value on exit from a routine regardless of the <code>ipar[1]</code> setting.</p>
2	<p>Contains warning messaging options:</p> <ul style="list-style-type: none"> <code>ipar[2]=-1</code> indicates that all warning messages are printed to the <code>MKL_Poisson_Library_log.txt</code> file in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. <code>ipar[2]=0</code> indicates that no warning messages will be printed. <code>ipar[2]=1</code> is the default value. It indicates that all warning messages are printed to the standard output device. <p>In case of warnings, the <code>stat</code> parameter contains a non-zero value on exit from a routine regardless of the <code>ipar[2]</code> setting.</p>
3 through 5	Internal parameters.
Parameters 6 through 11 are used only in the Cartesian case.	
6	<p>Takes this value:</p> <ul style="list-style-type: none"> 2, if <code>BCTYPE[0]='P'</code> 1, if <code>BCTYPE[0]='N'</code> 0, if <code>BCTYPE[0]='D'</code> -1, otherwise
7	<p>Takes this value:</p> <ul style="list-style-type: none"> 2, if <code>BCTYPE[1]='P'</code>

Index	Description
	<ul style="list-style-type: none"> • 1, if <code>BCtype[1]='N'</code> • 0, if <code>BCtype[1]='D'</code> • -1, otherwise
8	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if <code>BCtype[2]='P'</code> • 1, if <code>BCtype[2]='N'</code> • 0, if <code>BCtype[2]='D'</code> • -1, otherwise
9	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if <code>BCtype[3]='P'</code> • 1, if <code>BCtype[3]='N'</code> • 0, if <code>BCtype[3]='D'</code> • -1, otherwise
10	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if <code>BCtype[4]='P'</code> • 1, if <code>BCtype[4]='N'</code> • 0, if <code>BCtype[4]='D'</code> • -1, otherwise
11	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if <code>BCtype[5]='P'</code> • 1, if <code>BCtype[5]='N'</code> • 0, if <code>BCtype[5]='D'</code> • -1, otherwise
12	<p>Takes the value of</p> <ul style="list-style-type: none"> • <code>nx</code>, that is, the number of intervals along the x-axis, in the Cartesian case. • <code>np</code>, that is, the number of intervals along the ϕ-axis, in the spherical case.
13	<p>Takes the value of</p> <ul style="list-style-type: none"> • <code>ny</code>, that is, the number of intervals along the y-axis, in the Cartesian case • <code>nt</code>, that is, the number of intervals along the θ-axis, in the spherical case.
14	<p>Takes the value of <code>nz</code>, the number of intervals along the z-axis. This parameter is used only in the 3D case (Cartesian).</p>
15 through 22	<p>Internal parameters which define the internal partitioning of the <code>dpar/spar</code> array.</p>
<p>The values of <code>ipar[21]</code> - <code>ipar[119]</code> are assigned regardless of the dimension of the problem for the Cartesian solver or of whether the solver on a sphere is periodic.</p>	
23	<p>Contains message style options. Specifically:</p> <ul style="list-style-type: none"> • <code>ipar[21]=0</code> indicates that Poisson Solver routines prints the messages in Fortran-style notations. • <code>ipar[21]=1</code> (default) indicates that Poisson Solver routines prints the messages in C-style notations.

Index	Description
24	Contains the number of OpenMP threads to be used for computations in a multithreaded environment. The default value is 1 in the serial mode, and the result returned by the <code>mkl_get_max_threads</code> function otherwise.
25 through 28	Internal parameters which define the internal partitioning of the <i>dpar/spar</i> array.
25	Takes the value of <i>ipar</i> [18]+1, which specifies the internal partitioning of the <i>dpar/spar</i> array in the periodic Cartesian case.
26	Takes the value of <i>ipar</i> [23]+3* <i>ipar</i> [12]/4, which specifies the internal partitioning of the <i>dpar/spar</i> array in the periodic Cartesian case.
27	Takes the value of <i>ipar</i> [20]+1, which specifies the internal partitioning of the <i>dpar/spar</i> array in the periodic 3D Cartesian case.
28	Takes the value of <i>ipar</i> [25]+3* <i>ipar</i> [13]/4, which specifies the internal partitioning of the <i>dpar/spar</i> array in the periodic 3D Cartesian case.
29 through 39	Unused.
40 through 59	Contain the first twenty elements of the <i>ipar</i> array of the first Trigonometric Transform that the solver uses. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
60 through 79	Contain the first twenty elements of the <i>ipar</i> array of the second Trigonometric Transform that the 3D Cartesian and periodic spherical solvers use. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
80 through 99	Contain the first twenty elements of the <i>ipar</i> array of the third Trigonometric Transform that the solver uses in case of periodic boundary conditions along the <i>x</i> -axis. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
100 through 119	Contain the first twenty elements of the <i>ipar</i> array of the fourth Trigonometric Transform used by periodic spherical solvers and 3D Cartesian solvers with periodic boundary conditions along the <i>y</i> -axis. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
120 through 128	Internal parameters used by nonuniform 3D solvers.

NOTE

While you can declare the *ipar* array as `MKL_INT ipar[120]`, for future compatibility you should declare *ipar* as `MKL_INT ipar[128]`.

dpar and spar

Arrays *dpar* and *spar* are the same except in the data precision:

dpar Holds data needed for double-precision Fast Helmholtz Solver computations.

- For the Cartesian solver, double array of size $5 \cdot n_x/2 + 7$ in the 2D case or $5 \cdot (n_x + n_y)/2 + 9$ in the 3D case; initialized in the `d_init_Helmholtz_2D/d_init_Helmholtz_3D` and `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D` routines.
- For the spherical solver, double array of size $5 \cdot n_p/2 + n_t + 10$; initialized in the `d_init_sph_p/d_init_sph_np` and `d_commit_sph_p/d_commit_sph_np` routines.

spar

Holds data needed for single-precision Fast Helmholtz Solver computations.

- For the Cartesian solver, float array of size $5 \cdot n_x/2 + 7$ in the 2D case or $5 \cdot (n_x + n_y)/2 + 9$ in the 3D case; initialized in the `s_init_Helmholtz_2D/s_init_Helmholtz_3D` and `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D` routines.
- For the spherical solver, float array of size $5 \cdot n_p/2 + n_t + 10$; initialized in the `s_init_sph_p/s_init_sph_np` and `s_commit_sph_p/s_commit_sph_np` routines.

Because *dpar* and *spar* have similar elements in each position, the elements are described together in [Table "Elements of the dpar and spar Arrays"](#):

Elements of the dpar and spar Arrays

Index	Description
0	<p>In the Cartesian case, contains the length of the interval along the x-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_x in the x direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine.</p> <p>In the spherical case, contains the length of the interval along the ϕ-axis right after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine or the mesh size h_ϕ in the ϕ direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.</p>
1	<p>In the Cartesian case, contains the length of the interval along the y-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_y in the y direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine.</p> <p>In the spherical case, contains the length of the interval along the θ-axis right after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine or the mesh size h_θ in the θ direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.</p>
2	<p>In the Cartesian case, contains the length of the interval along the z-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_z in the z direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. In the Cartesian solver, this parameter is used only in the 3D case.</p> <p>In the spherical solver, contains the coordinate of the leftmost boundary along the θ-axis after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine.</p>
3	<p>Contains the value of the coefficient q after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np</code> routine.</p>

Index	Description
4	<p>Contains the tolerance parameter after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np</code> routine.</p> <ul style="list-style-type: none"> In the Cartesian case, this value is used only for the pure Neumann boundary conditions (<code>BCTYPE="NNNN"</code> in the 2D case; <code>BCTYPE="NNNNNN"</code> in the 3D case). This is a special case, because the right-hand side of the problem cannot be arbitrary if the coefficient q is zero. The Poisson Solver verifies that the classical solution exists (up to rounding errors) using this tolerance. In any case, the Poisson Solver computes the normal solution, that is, the solution that has the minimal Euclidean norm of residual. Nevertheless, the <code>?_Helmholtz_2D/?_Helmholtz_3D</code> routine informs you that the solution may not exist in a classical sense (up to rounding errors). In the spherical case, the value is used for the special case of a periodic problem on the entire sphere. This special case is similar to the Cartesian case with pure Neumann boundary conditions. Here the Poisson Solver computes the normal solution as well. The parameter is also used to detect whether the problem is periodic up to rounding errors. <p>The default value for this parameter is 1.0E-10 in case of double-precision computations or 1.0E-4 in case of single-precision computations. You can increase the value of the tolerance, for instance, to avoid the warnings that may appear.</p>
<code>ipar[15]-1</code> through <code>ipar[16]-1</code>	<p>In the Cartesian case, contain the spectrum of the one-dimensional (1D) problem along the x-axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine.</p> <p>In the spherical case, contains the spectrum of the 1D problem along the ϕ-axis after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.</p>
<code>ipar[17]-1</code> through <code>ipar[18]-1</code>	<p>In the Cartesian case, contain the spectrum of the 1D problem along the y-axis after a call to the <code>?_commit_Helmholtz_3D</code> routine. These elements are used only in the 3D case.</p> <p>In the spherical case, contains the spherical weights after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.</p>
<code>ipar[19]-1</code> through <code>ipar[20]-1</code>	<p>Take the values of the (staggered) sine/cosine in the mesh points:</p> <ul style="list-style-type: none"> along the x-axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine for a Cartesian solver along the ϕ-axis after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine for a spherical solver.
<code>ipar[21]-1</code> through <code>ipar[22]-1</code>	<p>Take the values of the (staggered) sine/cosine in the mesh points:</p> <ul style="list-style-type: none"> along the y-axis after a call to the <code>?_commit_Helmholtz_3D</code> routine for a Cartesian 3D solver along the ϕ-axis after a call to the <code>?_commit_sph_p</code> routine for a spherical periodic solver. <p>These elements are not used in the 2D Cartesian case and in the non-periodic spherical case.</p>
<code>ipar[25]-1</code> through <code>ipar[26]-1</code>	<p>Take the values of the (staggered) sine/cosine in the mesh points along the x-axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. These elements are used only in the periodic Cartesian case.</p>

Index	Description
<i>ipar</i> [27]-1 through <i>ipar</i> [28]	Take the values of the (staggered) sine/cosine in the mesh points along the x-axis after a call to the ?_commit_Helmholtz_3D routine. These elements are used only in the periodic 3D Cartesian case.

NOTE

You may define the array size depending upon the type of the problem to solve.

Caveat on Parameter Modifications

Flexibility of the Poisson Solver interface enables you to skip calls to the ?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np routine and to initialize the basic data structures explicitly in your code. You may also need to modify contents of the *ipar*, *dpar*, and *spar* arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or incorrect results. You can perform a basic check for correctness and consistency of parameters by calling the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine; however, this does not ensure the correct solution but only reduces the chance of errors or wrong results.

NOTE

To supply correct and consistent parameters to Poisson Solver routines, you should have considerable experience in using the Poisson Solver interface and good understanding of the solution process, as well as elements contained in the *ipar*, *spar*, and *dpar* arrays and dependencies between values of these elements.

In rare occurrences when you fail in tuning parameters for the Fast Helmholtz Solver, refer for technical support at <http://www.intel.com/software/products/support/>.

WARNING

The only way that ensures a proper solution of a Helmholtz problem is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of *ipar*, *dpar*, and *spar* arrays unless it is necessary.

Parameters That Define Boundary Conditions

Poisson Solver routines for the Cartesian solver use the following common parameters to define the boundary conditions.

Parameters to Define Boundary Conditions for the Cartesian Solver

Parameter	Description
<i>bd_ax</i>	<p>double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D and d_Helmholtz_2D/d_Helmholtz_3D,</p> <p>float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D and s_Helmholtz_2D/s_Helmholtz_3D.</p> <p>Contains values of the boundary condition on the leftmost boundary of the domain along the x-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is <i>ny</i>+1. Its contents depend on the boundary conditions as follows:

Parameter	Description
	<ul style="list-style-type: none"> Dirichlet boundary condition (value of <i>BCtype</i>[0] is 'D'): values of the function $G(ax, y_j)$, $j=0, \dots, ny$. Neumann boundary condition (value of <i>BCtype</i>[0] is 'N'): values of the function $g(ax, y_j)$, $j=0, \dots, ny$. <p>The value corresponding to the index j is placed in <i>bd_ax</i>[j].</p> <ul style="list-style-type: none"> 3D problem: the size of the array is $(ny+1)*(nz+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of <i>BCtype</i>[0] is 'D'): values of the function $G(ax, y_j, z_k)$, $j=0, \dots, ny, k=0, \dots, nz$. Neumann boundary condition (value of <i>BCtype</i>[0] is 'N'): the values of the function $g(ax, y_j, z_k)$, $j=0, \dots, ny, k=0, \dots, nz$. <p>The values are packed in the array so that the value corresponding to indices (j, k) is placed in <i>bd_ax</i>[$j+k*(ny+1)$].</p> <p>For periodic boundary conditions (the value of <i>BCtype</i>[0] is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<i>bd_bx</i>	<p>double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D and d_Helmholtz_2D/d_Helmholtz_3D,</p> <p>float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D and s_Helmholtz_2D/s_Helmholtz_3D.</p> <p>Contains values of the boundary condition on the rightmost boundary of the domain along the x-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is $ny+1$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of <i>BCtype</i>[1] is 'D'): values of the function $G(bx, y_j)$, $j=0, \dots, ny$. Neumann boundary condition (value of <i>BCtype</i>[1] is 'N'): values of the function $g(bx, y_j)$, $j=0, \dots, ny$. <p>The value corresponding to the index j is placed in <i>bd_bx</i>[j].</p> <ul style="list-style-type: none"> 3D problem: the size of the array is $(ny+1)*(nz+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of <i>BCtype</i>[1] is 'D'): values of the function $G(bx, y_j, z_k)$, $j=0, \dots, ny, k=0, \dots, nz$. Neumann boundary condition (value of <i>BCtype</i>[1] is 'N'): values of the function $g(bx, y_j, z_k)$, $j=0, \dots, ny, k=0, \dots, nz$. <p>The values are packed in the array so that the value corresponding to indices (j, k) is placed in <i>bd_bx</i>[$j+k*(ny+1)$].</p> <p>For periodic boundary conditions (the value of <i>BCtype</i>[1] is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<i>bd_ay</i>	<p>double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D and d_Helmholtz_2D/d_Helmholtz_3D,</p> <p>float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D and s_Helmholtz_2D/s_Helmholtz_3D.</p> <p>Contains values of the boundary condition on the leftmost boundary of the domain along the y-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is $nx+1$. Its contents depend on the boundary conditions as follows:

Parameter	Description
	<ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[2]</code> is 'D'): values of the function $G(x_i, ay)$, $i=0, \dots, nx$. Neumann boundary condition (value of <code>BCtype[2]</code> is 'N'): values of the function $g(x_i, ay)$, $i=0, \dots, nx$. <p>The value corresponding to the index i is placed in <code>bd_ay[i]</code>.</p> <ul style="list-style-type: none"> 3D problem: the size of the array is $(nx+1)*(nz+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[2]</code> is 'D'): values of the function $G(x_i, ay, z_k)$, $i=0, \dots, nx$, $k=0, \dots, nz$. Neumann boundary condition (value of <code>BCtype[2]</code> is 'N'): values of the function $g(x_i, ay, z_k)$, $i=0, \dots, nx$, $k=0, \dots, nz$. <p>The values are packed in the array so that the value corresponding to indices (i, k) is placed in <code>bd_ay[i+k*(nx+1)]</code>.</p> <p>For periodic boundary conditions (the value of <code>BCtype[2]</code> is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<code>bd_by</code>	<p>double* for <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code> and <code>d_Helmholtz_2D/d_Helmholtz_3D</code>,</p> <p>float* for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_2D/s_Helmholtz_3D</code>.</p> <p>Contains values of the boundary condition on the rightmost boundary of the domain along the y-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is $nx+1$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[3]</code> is 'D'): values of the function $G(x_i, by)$, $i=0, \dots, nx$. Neumann boundary condition (value of <code>BCtype[3]</code> is 'N'): values of the function $g(x_i, by)$, $i=0, \dots, nx$. <p>The value corresponding to the index i is placed in <code>bd_by[i]</code>.</p> 3D problem: the size of the array is $(nx+1)*(nz+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[3]</code> is 'D'): values of the function $G(x_i, by, z_k)$, $i=0, \dots, nx$, $k=0, \dots, nz$. Neumann boundary condition (value of <code>BCtype[3]</code> is 'N'): values of the function $g(x_i, by, z_k)$, $i=0, \dots, nx$, $k=0, \dots, nz$. <p>The values are packed in the array so that the value corresponding to indices (i, k) is placed in <code>bd_by[i+k*(nx+1)]</code>.</p> <p>For periodic boundary conditions (the value of <code>BCtype[3]</code> is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<code>bd_az</code>	<p>double* for <code>d_commit_Helmholtz_3D</code> and <code>d_Helmholtz_3D</code>,</p> <p>float* for <code>s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_3D</code>.</p> <p>Used only by <code>?_commit_Helmholtz_3D</code> and <code>?_Helmholtz_3D</code>. Contains values of the boundary condition on the leftmost boundary of the domain along the z-axis.</p> <p>The size of the array is $(nx+1)*(ny+1)$. Its contents depend on the boundary conditions as follows:</p> <ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[4]</code> is 'D'): values of the function $G(x_i, y_j, az)$, $i=0, \dots, nx$, $j=0, \dots, ny$.

Parameter	Description
	<ul style="list-style-type: none"> Neumann boundary condition (value of <i>BCtype</i>[4] is 'N'), values of the function $g(x_i, y_j, az)$, $i=0, \dots, nx, j=0, \dots, ny$. <p>The values are packed in the array so that the value corresponding to indices (i, j) is placed in $bd_az[i+j*(nx+1)]$.</p> <p>For periodic boundary conditions (the value of <i>BCtype</i>[4] is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<i>bd_bz</i>	<p>double* for <i>d_commit_Helmholtz_3D</i> and <i>d_Helmholtz_3D</i>, float* for <i>s_commit_Helmholtz_3D</i> and <i>s_Helmholtz_3D</i>.</p> <p>Used only by <i>?_commit_Helmholtz_3D</i> and <i>?_Helmholtz_3D</i>. Contains values of the boundary condition on the rightmost boundary of the domain along the z-axis.</p> <p>The size of the array is $(nx+1)*(ny+1)$. Its contents depend on the boundary conditions as follows:</p> <ul style="list-style-type: none"> Dirichlet boundary condition (value of <i>BCtype</i>[5] is 'D'): values of the function $G(x_i, y_j, bz)$, $i=0, \dots, nx, j=0, \dots, ny$. Neumann boundary condition (value of <i>BCtype</i>[5] is 'N'): values of the function $g(x_i, y_j, bz)$, $i=0, \dots, nx, j=0, \dots, ny$. <p>The values are packed in the array so that the value corresponding to indices (i, j) is placed in $bd_bz[i+j*(nx+1)]$.</p> <p>For periodic boundary conditions (the value of <i>BCtype</i>[5] is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>

See Also

[?_commit_Helmholtz_2D/?_commit_Helmholtz_3D](#)

[?_Helmholtz_2D/?_Helmholtz_3D](#)

Poisson Solver Implementation Details

Several aspects of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver interface are platform-specific and language-specific. To promote portability of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver interface across platforms and ease of use across different languages, Intel® oneAPI Math Kernel Library (oneMKL) provides you with the Poisson Solver language-specific header file to include in your code:

- mkl_poisson.h*, to be used together with *mkl_dfti.h*.

NOTE

- Use of the Intel® oneAPI Math Kernel Library (oneMKL) Poisson Solver software without including the above language-specific header files is not supported.

Header File

The header file defines the function prototypes for the Cartesian and spherical solver available in specific function descriptions.

Nonlinear Optimization Problem Solvers

Intel® oneAPI Math Kernel Library (oneMKL) provides tools for solving nonlinear least squares problems using the Trust-Region (TR) algorithms. The general nonlinear solver workflow and naming conventions are described here:

- [Nonlinear Solver Organization and Implementation](#)
- [Nonlinear Solver Routine Naming Conventions](#)

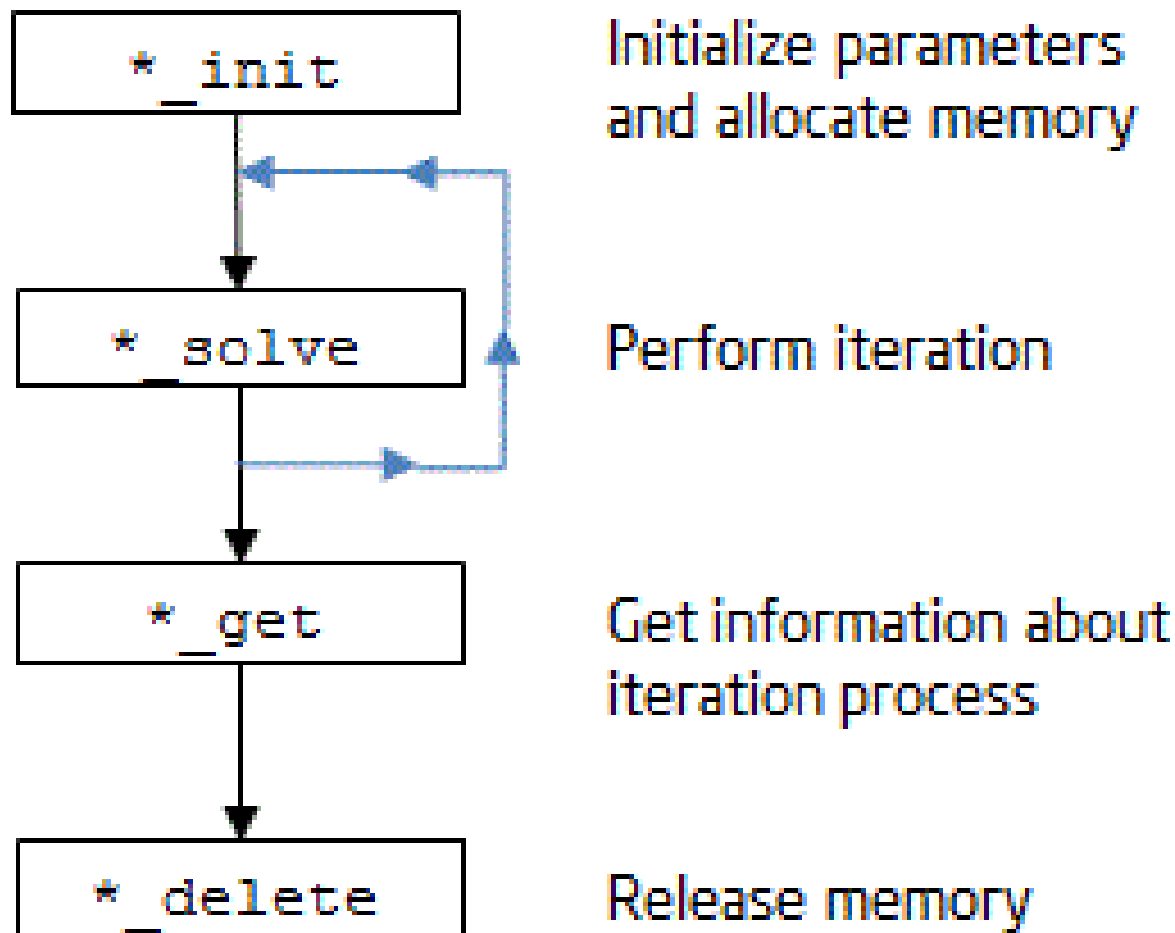
The solver routines are grouped according to their purpose as follows:

- [Nonlinear Least Squares Problem without Constraints](#)
- [Nonlinear Least Squares Problem with Linear \(Boundary\) Constraints](#)
- [Jacobian Matrix Calculation Routines](#)

For more information on the key concepts required to understand the use of the Intel® oneAPI Math Kernel Library (oneMKL) nonlinear least squares problem solver routines, see [[Conn00](#)].

Nonlinear Solver Organization and Implementation

The Intel® oneAPI Math Kernel Library (oneMKL) solver routines for nonlinear least squares problems use reverse communication interfaces (RCI). That means you need to provide the solver with information required for the iteration process, for example, the corresponding Jacobian matrix, or values of the objective function. RCI removes the dependency of the solver on specific implementation of the operations. However, it does require that you organize a computational loop.

`__border__top`**Typical order for invoking RCI solver routines**

The nonlinear least squares problem solver routines, or Trust-Region (TR) solvers, are implemented with threading support. You can manage the threads using [Threading Control Functions](#). The TR solvers use BLAS and LAPACK routines, and offer the same parallelism as those domains. The `?jacobi` and `?jacobix` routines of Jacobi matrix calculations are parallel. These routines (`?jacobi` and `?jacobix`) make calls to the user-supplied functions with different `x` parameters for multiple threads.

Memory Allocation and Handles

To make the TR solver routines easy to use, you are not required to allocate temporary working storage. The solver allocates all temporary memory internally. To allow multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using a data object called a handle. Each TR solver routine creates, uses, or deletes a handle. The handle datatype definition can be found in `mk1.h` (or `mk1_rci.h`/`mk1_rci_win.h`).

Include one of the mentioned headers and declare the handle as :

```
TRNSP_HANDLE_t handle;
```


or

```
_TRNSPBC_HANDLE_t handle;
```

The first declaration is used for nonlinear least squares problems without boundary constraints, and the second is used for nonlinear least squares problems with boundary constraints.

Product and Performance Information

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

Notice revision #20201201

Nonlinear Solver Routine Naming Conventions

The TR routine names have the following structure:

```
<character><name>_<action>( )
```

where

- **<character>** indicates the data type:

s	float
d	double

- **<name>** indicates the task type:

trnlsp	nonlinear least squares problem without constraints
trnlspbc	nonlinear least squares problem with boundary constraints
jacobi	computation of the Jacobian matrix using central differences

- **<action>** indicates an action on the task:

init	initializes the solver
check	checks correctness of the input parameters
solve	solves the problem
get	retrieves the number of iterations, the stop criterion, the initial residual, and the final residual
delete	releases the allocated data

Nonlinear Least Squares Problem without Constraints

The nonlinear least squares problem without constraints can be described as follows:

$$\min_{x \in R^n} \|F(x)\|_2^2 = \min_{x \in R^n} \|y - f(x)\|_2^2, y \in R^m, x \in R^n, f: R^n \rightarrow R^m, m \geq n,$$

where

$F(x) : R^n \rightarrow R^m$ is a twice differentiable function in R^n .

Solving a nonlinear least squares problem means searching for the best approximation to the vector y with the model function $\hat{f}_i(x)$ and nonlinear variables x . The best approximation means that the sum of squares of residuals $y_i - \hat{f}_i(x)$ is the minimum.

See usage examples in the `examples\c\nonlinear_solvers` folder of your Intel® oneAPI Math Kernel Library (oneMKL) directory. Specifically, see `ex_nlsqp_f.fex_nlsqp_c.c`.

RCI TR Routines

Routine Name	Operation
<code>?trnlsqp_init</code>	Initializes the solver.
<code>?trnlsqp_check</code>	Checks correctness of the input parameters.
<code>?trnlsqp_solve</code>	Solves a nonlinear least squares problem using the Trust-Region algorithm.
<code>?trnlsqp_get</code>	Retrieves the number of iterations, stop criterion, initial residual, and final residual.
<code>?trnlsqp_delete</code>	Releases allocated data.

?trnlsqp_init

Initializes the solver of a nonlinear least squares problem.

Syntax

```
MKL_INT strnlsqp_init (_TRNSP_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m,
const float* x, const float* eps, const MKL_INT* iter1, const MKL_INT* iter2, const
float* rs);
```

```
MKL_INT dtrnlsqp_init (_TRNSP_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m,
const double* x, const double* eps, const MKL_INT* iter1, const MKL_INT* iter2, const
double* rs);
```

Include Files

- `mk1.h`

Description

The `?trnlsqp_init` routine initializes the solver.

After initialization, all subsequent invocations of the `?trnlsqp_solve` routine should use the values of the handle returned by `?trnlsqp_init`. This handle stores internal data, including pointers to the arrays `x` and `eps`. It is important to not move or deallocate these arrays until after calling the `?trnlsqp_delete` routine.

The `eps` array contains a number indicating the stopping criteria:

<code>eps</code> Value	Description
0	$\Delta < eps[0]$
1	$ F(x) _2 < eps[1]$
2	The Jacobian matrix is singular. $ J(x)_{[m*(j-1)...m*j-1]} _2 < eps[2], j = 1, \dots, n$
3	$ s _2 < eps[3]$
4	$ F(x) _2 - F(x) - J(x)s _2 < eps[4]$

<i>eps</i> Value	Description
5	The trial step precision. If $eps[5] = 0$, then the trial step meets the required precision ($\leq 1.0 \cdot 10^{-10}$).

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

<i>n</i>	Length of x .
<i>m</i>	Length of $F(x)$.
<i>x</i>	Array of size n . Initial guess. A reference to this array is stored in <i>handle</i> for later use and modification by <code>?trnlsp_solve</code> .
<i>eps</i>	Array of size 6; contains stopping criteria. See the values in the Description section. A reference to this array is stored in <i>handle</i> for later use by <code>?trnlsp_solve</code> .
<i>iter1</i>	Specifies the maximum number of iterations.
<i>iter2</i>	Specifies the maximum number of iterations of trial step calculation.
<i>rs</i>	Definition of initial size of the trust region (boundary of the trial step). The recommend minimum value is 0.1, and the recommended maximum value is 100.0. Based on your knowledge of the objective function and initial guess you can increase or decrease the initial trust region. It can influence the iteration process, for example, the direction of the iteration process and the number of iterations. If you set <i>rs</i> to 0.0, the solver uses the default value, which is 100.0.

Output Parameters

<i>handle</i>	Type <code>_TRNSP_HANDLE_t</code> .
<i>res</i>	Indicates task completion status. <ul style="list-style-type: none"> • <i>res</i> = <code>TR_SUCCESS</code> - the routine completed the task normally. • <i>res</i> = <code>TR_INVALID_OPTION</code> - there was an error in the input parameters. • <i>res</i> = <code>TR_OUT_OF_MEMORY</code> - there was a memory error. <p><code>TR_SUCCESS</code>, <code>TR_INVALID_OPTION</code>, and <code>TR_OUT_OF_MEMORY</code> are defined in the <code>mkl_rci.h</code> include file.</p>

See Also

[?trnlsp_solve](#)

?trnlsp_check

Checks the correctness of *handle* and arrays containing Jacobian matrix, objective function, and stopping criteria.

Syntax

```
MKL_INT strnlsp_check (_TRNSP_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m,
const float* fjac, const float* fvec, const float* eps, MKL_INT* info);
```

```
MKL_INT dtrnlsp_check (_TRNSP_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m,
const double* fjac, const double* fvec, const double* eps, MKL_INT* info);
```

Include Files

- mkl.h

Description

The ?trnlsp_check routine checks the arrays passed into the solver as input parameters. If an array contains any INF or NaN values, the routine sets the flag in output array *info* (see the description of the values returned in the Output Parameters section for the *info* array).

Input Parameters

<i>handle</i>	Type <code>_TRNSP_HANDLE_t</code> .
<i>n</i>	Length of <i>x</i> .
<i>m</i>	Length of <i>F(x)</i> .
<i>fjac</i>	Array of size <i>m</i> by <i>n</i> . Contains the Jacobian matrix of the function.
<i>fvec</i>	Array of size <i>m</i> . Contains the function values at <i>x</i> , where $fvec[i] = (y_i - \hat{f}_i(x))$.
<i>eps</i>	Array of size 6; contains stopping criteria. See the values in the Description section of the ?trnlsp_init.

Output Parameters

info Array of size 6.
Results of input parameter checking:

Parameter	Used for	Value	Description
<i>info</i> [0]	Flags for <i>handle</i>	0	The <i>handle</i> is valid.
		1	The <i>handle</i> is not allocated.
<i>info</i> [1]	Flags for <i>fjac</i>	0	The <i>fjac</i> array is valid.
		1	The <i>fjac</i> array is not allocated
		2	The <i>fjac</i> array contains NaN.
		3	The <i>fjac</i> array contains Inf.

Parameter	Used for	Value	Description
<i>info</i> [2]	Flags for <i>fvec</i>	0	The <i>fvec</i> array is valid.
		1	The <i>fvec</i> array is not allocated
		2	The <i>fvec</i> array contains NaN.
		3	The <i>fvec</i> array contains Inf.
<i>info</i> [3]	Flags for <i>eps</i>	0	The <i>eps</i> array is valid.
		1	The <i>eps</i> array is not allocated
		2	The <i>eps</i> array contains NaN.
		3	The <i>eps</i> array contains Inf.
		4	The <i>eps</i> array contains a value less than or equal to zero.

res

Information about completion of the task.

res = TR_SUCCESS - the routine completed the task normally.

TR_SUCCESS is defined in the `mkl_rci.h` include file.

?trnlsp_solve

Solves a nonlinear least squares problem using the TR algorithm.

Syntax

```
MKL_INT strnlsp_solve (_TRNSP_HANDLE_t* handle, float* fvec, float* fjac, MKL_INT* RCI_Request);
```

```
MKL_INT dtrnlsp_solve (_TRNSP_HANDLE_t* handle, double* fvec, double* fjac, MKL_INT* RCI_Request);
```

Include Files

- `mkl.h`

Description

The ?trnlsp_solve routine uses the TR algorithm to solve nonlinear least squares problems.

The problem is stated as follows:

$$\min_{x \in R^n} \|F(x)\|_2^2 = \min_{x \in R^n} \|y - f(x)\|_2^2, y \in R^m, x \in R^n, f: R^n \rightarrow R^m, m \geq n,$$

where

- $F(x): R^n \rightarrow R^m$
- $m \geq n$

From a current point $x_{current}$, the algorithm uses the trust-region approach:

$$\min_{x \in \mathbb{R}^n} \|F(x_{\text{current}}) + J(x_{\text{current}})(x_{\text{new}} - x_{\text{current}})\|_2^2 \quad \text{subject to } \|x_{\text{new}} - x_{\text{current}}\| \leq \Delta_{\text{current}}$$

to get $x_{\text{new}} = x_{\text{current}} + s$ that satisfies

$$\min_{x \in \mathbb{R}^n} \|J^T(x)J(x)s + J^T F(x)\|_2^2$$

where

- $J(x)$ is the Jacobian matrix
- s is the trial step
- $\|s\|_2 \leq \Delta_{\text{current}}$
- Δ is the trust-region area.

The `RCI_Request` parameter provides additional information:

<code>RCI_Request</code> Value	Description
2	Request to calculate the Jacobian matrix and put the result into <code>fjac</code>
1	Request to recalculate the function at vector <code>x</code> and put the result into <code>fvec</code>
0	One successful iteration step on the current trust-region radius (that does not mean that the value of <code>x</code> has changed)
-1	The algorithm has exceeded the maximum number of iterations
-2	$\Delta < \text{eps}[0]$
-3	$\ F(x)\ _2 < \text{eps}[1]$
-4	The Jacobian matrix is singular. $\ J(x)_{[m*(j-1)...m*j-1]}\ _2 < \text{eps}[2], j = 1, \dots, n$
-5	$\ s\ _2 < \text{eps}[3]$
-6	$\ F(x)\ _2 - \ F(x) - J(x)s\ _2 < \text{eps}[4]$

NOTE

If it is possible to combine computations of the function and the jacobian (`RCI_Request` = 1 and 2), you can do that and provide both updated values for `fvec` and `fjac` as fulfillment of `RCI_Request` = 1 (and do nothing for `RCI_Request` = 2).

Input Parameters

<code>handle</code>	Type <code>_TRNSP_HANDLE_t</code> .
<code>fvec</code>	Array of size m . Contains the function values at <code>x</code> , where $fvec[i] = (y_i - f_i(x))$.

fjac Array of size m by n . Contains the Jacobian matrix of the function.

Output Parameters

fvec Array of size m . Updated function evaluated at x .

RCI_Request Informs about the task stage.

See the Description section for the parameter values and their meaning.

res Indicates the task completion.

res = TR_SUCCESS - the routine completed the task normally.

TR_SUCCESS is defined in the `mkl_rci.h` include file.

?trnlsp_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

```
MKL_INT strnlsp_get(_TRNSP_HANDLE_t* handle, MKL_INT* iter, MKL_INT* st_cr, float* r1, float* r2);
```

```
MKL_INT dtrnlsp_get (_TRNSP_HANDLE_t* handle, MKL_INT* iter, MKL_INT* st_cr, double* r1, double* r2);
```

Include Files

- `mkl.h`

Description

The routine retrieves the current number of iterations, the stop criterion, the initial residual, and final residual.

The initial residual is the value of the functional $(||y - f(x)||)$ of the initial x values provided by the user.

The final residual is the value of the functional $(||y - f(x)||)$ of the final x resulting from the algorithm operation.

The *st_cr* parameter contains a number indicating the stop criterion:

<i>st_cr</i> Value	Description
1	The algorithm has exceeded the maximum number of iterations
2	$\Delta < eps[0]$
3	$ F(x) _2 < eps[1]$
4	The Jacobian matrix is singular. $ J(x)_{[m*(j-1)...m*j-1]} _2 < eps[2], j = 1, \dots, n$
5	$ s _2 < eps[3]$
6	$ F(x) _2 - F(x) - J(x)s _2 < eps[4]$

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

handle Type `_TRNSP_HANDLE_t`.

Output Parameters

iter Contains the current number of iterations.

st_cr Contains the stop criterion.
See the Description section for the parameter values and their meanings.

r1 Contains the residual, $(||y - f(x)||)$ given the initial x .

r2 Contains the final residual, that is, the value of the functional $(||y - f(x)||)$ of the final x resulting from the algorithm operation.

res Indicates the task completion.
res = `TR_SUCCESS` - the routine completed the task normally.
`TR_SUCCESS` is defined in the `mk1_rci.h` include file.

?trnlsp_delete

Releases allocated data.

Syntax

```
MKL_INT strnlsp_delete(_TRNSP_HANDLE_t* handle);
MKL_INT dtrnlsp_delete(_TRNSP_HANDLE_t* handle);
```

Include Files

- `mk1.h`

Description

The `?trnlsp_delete` routine releases all memory allocated for the handle. Only after calling this routine is it safe for the user to move or deallocate the memory referenced by x and eps .

This routine flags memory as not used, but to actually release all memory you must call the support function [mk1_free_buffers](#).

Input Parameters

handle Type `_TRNSP_HANDLE_t`.

Output Parameters

res Indicates the task completion.
res = `TR_SUCCESS` means the routine completed the task normally.
`TR_SUCCESS` is defined in the `mk1_rci.h` include file.

Nonlinear Least Squares Problem with Linear (Bound) Constraints

The nonlinear least squares problem with linear bound constraints is very similar to the [nonlinear least squares problem without constraints](#) but it has the following constraints:

$$l_i \leq x_i \leq u_i, i = 1, \dots, n, \quad l, u \in R^n.$$

See usage examples in the `examples\c\nonlinear_solvers` folder of your Intel® oneAPI Math Kernel Library (oneMKL) directory. Specifically, see `ex_nlsqp_bc_c.c`.

NOTE There are two options for handling the boundary constraints enabled through the parameter array, see the description of [eps\[4\]](#)

RCI TR Routines for Problem with Bound Constraints

Routine Name	Operation
<code>?trnlsdbc_init</code>	Initializes the solver.
<code>?trnlsdbc_check</code>	Checks correctness of the input parameters.
<code>?trnlsdbc_solve</code>	Solves a nonlinear least squares problem using RCI and the Trust-Region algorithm.
<code>?trnlsdbc_get</code>	Retrieves the number of iterations, stop criterion, initial residual, and final residual.
<code>?trnlsdbc_delete</code>	Releases allocated data.

`?trnlsdbc_init`

Initializes the solver of nonlinear least squares problem with linear (boundary) constraints.

Syntax

```
MKL_INT strnlsdbc_init (_TRNSPBC_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m,
const float* x, const float* LW, const float* UP, const float* eps, const MKL_INT*
iter1, const MKL_INT* iter2, const float* rs);
```

```
MKL_INT dtrnlsdbc_init (_TRNSPBC_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m,
const double* x, const double* LW, const double* UP, const double* eps, const MKL_INT*
iter1, const MKL_INT* iter2, const double* rs);
```

Description

The `?trnlsdbc_init` routine initializes the solver.

After initialization, all subsequent invocations of the `?trnlsdbc_solve` routine should use the values of the handle returned by `?trnlsdbc_init`. This handle stores internal data, including pointers to the arrays `x`, `LW`, `UP`, and `eps`. It is important to not move or deallocate these arrays until after calling the `?trnlsdbc_delete` routine.

The `eps` array contains a number indicating the stopping criteria:

<i>eps</i> Value	Description
0	$\Delta < eps[0]$
1	$ F(x) _2 < eps[1]$
2	The Jacobian matrix is singular. $ J(x)_{[m*(j-1)...m*j-1]} _2 < eps[2], j = 1, \dots, n$
3	$ s _2 < eps[3]$
4	$ F(x) _2 - F(x) - J(x)s _2 < eps[4] $
<p>NOTE</p> <p>If $eps[4] > 0$, an extra scaling is applied to 's' after it has been selected, to ensure that it does not leave the specified domain, but scales it down to not cross the boundary. This preserves the solution inside the boundary, but may result in getting stuck in a local minimum on the boundary and exiting early due to this stopping criteria</p> <p>If $eps[4] < 0$, extra scaling is not applied, which may result in the solution, x, leaving the domain. If this occurs, try starting over with a different initial condition.</p>	
5	The trial step precision. If $eps[5] = 0$, then the trial step meets the required precision ($\leq 1.0 \cdot 10^{-10}$).

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

n	Length of x .
m	Length of $F(x)$.
x	Array of size n . Initial guess. A reference to this array is stored in handle for later use and modification by <code>?trnlspsc_solve</code> .
LW	Array of size n . Contains low bounds for x ($lw_i < x_i$). A reference to this array is stored in handle for later use by <code>?trnlspsc_solve</code> .
UP	Array of size n . Contains upper bounds for x ($up_i > x_i$). A reference to this array is stored in handle for later use by <code>?trnlspsc_solve</code> .
eps	Array of size 6; contains stopping criteria. See the values in the Description section. A reference to this array is stored in handle for later use by <code>?trnlspsc_solve</code> .

<code>iter1</code>	Specifies the maximum number of iterations.
<code>iter2</code>	Specifies the maximum number of iterations of trial step calculation.
<code>rs</code>	Definition of initial size of the trust region (boundary of the trial step). The recommended minimum value is 0.1, and the recommended maximum value is 100.0. Based on your knowledge of the objective function and initial guess you can increase or decrease the initial trust region. It can influence the iteration process, for example, the direction of the iteration process and the number of iterations. If you set <code>rs</code> to 0.0, the solver uses the default value, which is 100.0.

Output Parameters

<code>handle</code>	Type <code>_TRNSPBC_HANDLE_t</code> .
<code>res</code>	<p>Informs about the task completion.</p> <ul style="list-style-type: none"> <code>res = TR_SUCCESS</code> - the routine completed the task normally. <code>res = TR_INVALID_OPTION</code> - there was an error in the input parameters. <code>res = TR_OUT_OF_MEMORY</code> - there was a memory error. <p><code>TR_SUCCESS</code>, <code>TR_INVALID_OPTION</code>, and <code>TR_OUT_OF_MEMORY</code> are defined in the <code>mkl_rci.h</code> include file.</p>

?trnlsdbc_check

Checks the correctness of `handle` and arrays containing Jacobian matrix, objective function, lower and upper bounds, and stopping criteria.

Syntax

```
MKL_INT strnlsdbc_check (_TRNSPBC_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m,
const float* fjac, const float* fvec, const float* LW, const float* UP, const float*
eps, MKL_INT* info);
```

```
MKL_INT dtrnlsdbc_check (_TRNSPBC_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m,
const double* fjac, const double* fvec, const double* LW, const double* UP, const
double* eps, MKL_INT* info);
```

Include Files

- `mkl.h`

Description

The `?trnlsdbc_check` routine checks the arrays passed into the solver as input parameters. If an array contains any INF or NaN values, the routine sets the flag in output array `info` (see the description of the values returned in the Output Parameters section for the `info` array).

Input Parameters

<code>handle</code>	Type <code>_TRNSPBC_HANDLE_t</code> .
<code>n</code>	Length of x .
<code>m</code>	Length of $F(x)$.

<i>fjac</i>	Array of size m by n . Contains the Jacobian matrix of the function.
<i>fvec</i>	Array of size m . Contains the function values at x , where $fvec[i] = (y_i - f_i(x))$.
<i>LW</i>	Array of size n . Contains low bounds for x ($lw_i < x_i$).
<i>UP</i>	Array of size n . Contains upper bounds for x ($up_i > x_i$).
<i>eps</i>	Array of size 6; contains stopping criteria. See the values in the Description section of the <code>?trnlspsc_init</code> .

Output Parameters

<i>info</i>	Array of size 6. Results of input parameter checking:
-------------	--

Parameter	Used for	Value	Description
<i>info</i> [0]	Flags for <i>handle</i>	0	The handle is valid.
		1	The handle is not allocated.
<i>info</i> [1]	Flags for <i>fjac</i>	0	The <i>fjac</i> array is valid.
		1	The <i>fjac</i> array is not allocated
		2	The <i>fjac</i> array contains NaN.
		3	The <i>fjac</i> array contains Inf.
<i>info</i> [2]	Flags for <i>fvec</i>	0	The <i>fvec</i> array is valid.
		1	The <i>fvec</i> array is not allocated
		2	The <i>fvec</i> array contains NaN.
		3	The <i>fvec</i> array contains Inf.
<i>info</i> [3]	Flags for <i>LW</i>	0	The <i>LW</i> array is valid.
		1	The <i>LW</i> array is not allocated
		2	The <i>LW</i> array contains NaN.
		3	The <i>LW</i> array contains Inf.
		4	The lower bound is greater than the upper bound.
<i>info</i> [4]	Flags for <i>up</i>	0	The <i>up</i> array is valid.
		1	The <i>up</i> array is not allocated

Parameter	Used for	Value	Description
		2	The <i>up</i> array contains NaN.
		3	The <i>up</i> array contains Inf.
		4	The upper bound is less than the lower bound.
<i>info</i> [5]	Flags for <i>eps</i>	0	The <i>eps</i> array is valid.
		1	The <i>eps</i> array is not allocated
		2	The <i>eps</i> array contains NaN.
		3	The <i>eps</i> array contains Inf.
		4	The <i>eps</i> array contains a value less than or equal to zero.

res Information about completion of the task.
res = TR_SUCCESS - the routine completed the task normally.
TR_SUCCESS is defined in the `mkl_rci.h` include file.

?trnlspbc_solve

Solves a nonlinear least squares problem with linear (bound) constraints using the Trust-Region algorithm.

Syntax

```
MKL_INT strnlspbc_solve (_TRNSPBC_HANDLE_t* handle, float* fvec, float* fjac, MKL_INT* RCI_Request);  
  
MKL_INT dtrnlspbc_solve (_TRNSPBC_HANDLE_t* handle, double* fvec, double* fjac, MKL_INT* RCI_Request);
```

Include Files

- `mkl.h`

Description

The ?trnlspbc_solve routine, based on RCI, uses the Trust-Region algorithm to solve nonlinear least squares problems with linear (bound) constraints. The problem is stated as follows:

$$\min_{x \in R^n} \|F(x)\|_2^2 = \min_{x \in R^n} \|y - f(x)\|_2^2, y \in R^m, x \in R^n, f: R^n \rightarrow R^m, m \geq n$$

where

$$l_i \leq x_i \leq u_i$$
$$i = 1, \dots, n.$$

The *RCI_Request* parameter provides additional information:

<i>RCI_Request</i> Value	Description
2	Request to calculate the Jacobian matrix and put the result into <i>fjac</i>
1	Request to recalculate the function at vector <i>x</i> and put the result into <i>fvec</i>
0	One successful iteration step on the current trust-region radius (that does not mean that the value of <i>x</i> has changed)
-1	The algorithm has exceeded the maximum number of iterations
-2	$\Delta < eps[0]$
-3	$ F(x) _2 < eps[1]$
-4	The Jacobian matrix is singular. $ J(x)_{[m*(j-1)...m*j-1]} _2 < eps[2], j = 1, \dots, n$
-5	$ s _2 < eps[3]$
-6	$ F(x) _2 - F(x) - J(x)s _2 < eps[4] $

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

<i>handle</i>	Type <code>_TRNSPBC_HANDLE_t</code> .
<i>fvec</i>	Array of size <i>m</i> . Contains the function values at <i>x</i> , where $fvec[i] = (y_i - f_i(x))$.
<i>fjac</i>	Array of size <i>m</i> by <i>n</i> . Contains the Jacobian matrix of the function.

Output Parameters

<i>fvec</i>	Array of size <i>m</i> . Updated function evaluated at <i>x</i> .
<i>RCI_Request</i>	<p>Informs about the task stage.</p> <p>See the Description section for the parameter values and their meaning.</p>
<i>res</i>	<p>Informs about the task completion.</p> <p><code>res = TR_SUCCESS</code> means the routine completed the task normally.</p> <p><code>TR_SUCCESS</code> is defined in the <code>mkl_rci.h</code> include file.</p>

?trnlspsc_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

```
MKL_INT strnlspsc_get (_TRNSPBC_HANDLE_t* handle, MKL_INT* iter, MKL_INT* st_cr, float* r1, float* r2);
```

```
MKL_INT dtrnlspsc_get (_TRNSPBC_HANDLE_t* handle, MKL_INT* iter, MKL_INT* st_cr, double* r1, double* r2);
```

Include Files

- mkl.h

Description

The routine retrieves the current number of iterations, the stop criterion, the initial residual, and final residual.

The *st_cr* parameter contains a number indicating the stop criterion:

<i>st_cr</i> Value	Description
1	The algorithm has exceeded the maximum number of iterations
2	$\Delta < eps[0]$
3	$ F(x) _2 < eps[1]$
4	The Jacobian matrix is singular. $ J(x)_{[m*(j-1)...m*j-1]} _2 < eps[2], j = 1, \dots, n$
5	$ s _2 < eps[3]$
6	$ F(x) _2 - F(x) - J(x)s _2 < eps[4]$

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

handle Type `_TRNSPBC_HANDLE_t`.

Output Parameters

iter Contains the current number of iterations.

st_cr Contains the stop criterion.
See the Description section for the parameter values and their meanings.

r1 Contains the residual, $(||y - f(x)||)$ given the initial x .

r2 Contains the final residual, that is, the value of the function $(||y - f(x)||)$ of the final x resulting from the algorithm operation.

res Informs about the task completion.
res = `TR_SUCCESS` - the routine completed the task normally.

`TR_SUCCESS` is defined in the `mkl_rci.h` include file.

?trnlspsc_delete

Releases allocated data.

Syntax

```
MKL_INT strnlspsc_delete (_TRNSPBC_HANDLE_t* handle);
MKL_INT dtrnlspsc_delete (_TRNSPBC_HANDLE_t* handle);
```

Include Files

- `mkl.h`

Description

The `?trnlspsc_delete` routine releases all memory allocated for the handle. Only after calling this routine is it safe for the user to move or deallocate the memory referenced by *x*, *LW*, *UP*, and *eps*.

NOTE

This routine flags memory as not used, but to actually release all memory you must call the support function [mkl_free_buffers](#).

Input Parameters

handle Type `_TRNSPBC_HANDLE_t`.

Output Parameters

res Informs about the task completion.

res = `TR_SUCCESS` means the routine completed the task normally.

`TR_SUCCESS` is defined in the `mkl_rci.h` include file.

Jacobian Matrix Calculation Routines

This section describes routines that compute the Jacobian matrix using the central difference algorithm. Jacobian matrix calculation is required to solve a nonlinear least squares problem and systems of nonlinear equations (with or without linear bound constraints). Routines for calculation of the Jacobian matrix have the "Black-Box" interfaces, where you pass the objective function via parameters. Your objective function must have a fixed interface.

Jacobian Matrix Calculation Routines

Routine Name	Operation
<code>?jacobi_init</code>	Initializes the solver.
<code>?jacobi_solve</code>	Computes the Jacobian matrix of the function on the basis of RCI using the central difference algorithm.
<code>?jacobi_delete</code>	Removes data.

Routine Name	Operation
<code>?jacobi</code>	Computes the Jacobian matrix of the <code>fcn</code> function using the central difference algorithm.
<code>?jacobix</code>	Presents an alternative interface for the <code>?jacobi</code> function enabling you to pass additional data into the objective function.

`?jacobi_init`

Initializes the solver for Jacobian calculations.

Syntax

```
MKL_INT sjacobi_init (_JACOBIHANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const float* x, const float* fjac, const float* eps);
```

```
MKL_INT djacobi_init (_JACOBIHANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const double* x, const double* fjac, const double* eps);
```

Include Files

- `mkl.h`

Description

The routine initializes the solver.

Input Parameters

<code>n</code>	Length of <code>x</code> .
<code>m</code>	Length of <code>F</code> .
<code>x</code>	Array of size <code>n</code> . Vector, at which the function is evaluated. A reference to this array is stored in <code>handle</code> for later use and modification by <code>?jacobi_solve</code> .
<code>eps</code>	Precision of the Jacobian matrix calculation.
<code>fjac</code>	Array of size <code>m</code> by <code>n</code> . Contains the Jacobian matrix of the function. A reference to this array is stored in <code>handle</code> for later use and modification by <code>?jacobi_solve</code> .

Output Parameters

<code>handle</code>	Data object of the <code>_JACOBIHANDLE_t</code> type. Stores internal data, including pointers to the user-provided arrays <code>x</code> and <code>fjac</code> . It is important that the user does not move or deallocate these arrays until after calling the <code>?jacobi_delete</code> routine.
<code>res</code>	Indicates task completion status. <ul style="list-style-type: none"> • <code>res = TR_SUCCESS</code> - the routine completed the task normally. • <code>res = TR_INVALID_OPTION</code> - there was an error in the input parameters. • <code>res = TR_OUT_OF_MEMORY</code> - there was a memory error.

TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.h include file.

?jacobi_solve

Computes the Jacobian matrix of the function using RCI and the central difference algorithm.

Syntax

```
MKL_INT sjacobi_solve (_JACOBI_MATRIX_HANDLE_t* handle, float* f1, float* f2, MKL_INT* RCI_Request);
```

```
MKL_INT djacobi_solve (_JACOBI_MATRIX_HANDLE_t* handle, double* f1, double* f2, MKL_INT* RCI_Request);
```

Include Files

- mkl.h

Description

The ?jacobi_solve routine computes the Jacobian matrix of the function using RCI and the central difference algorithm.

See usage examples in the examples\solverc\source folder of your Intel® oneAPI Math Kernel Library (oneMKL) directory. Specifically, see sjacobi_rci_c.c and djacobi_rci_c.c.

Input Parameters

<i>handle</i>	Type <code>_JACOBI_MATRIX_HANDLE_t</code> .
<i>RCI_Request</i>	Set to 0 before the first call to ?jacobi_solve.

Output Parameters

<i>f1</i>	Contains the updated function values at $x + eps$.
<i>f2</i>	Array of size m . Contains the updated function values at $x - eps$.
<i>RCI_Request</i>	Provides information about the task completion. When equal to 0, the task has completed successfully. <i>RCI_Request</i> = 1 indicates that you should compute the function values at the current x point and put the results into <i>f1</i> . <i>RCI_Request</i> = 2 indicates that you should compute the function values at the current x point and put the results into <i>f2</i> .
<i>res</i>	Indicates the task completion status. <ul style="list-style-type: none"> • <i>res</i> = TR_SUCCESS - the routine completed the task normally. • <i>res</i> = TR_INVALID_OPTION - there was an error in the input parameters. TR_SUCCESS and TR_INVALID_OPTION are defined in the mkl_rci.h include file.

See Also

?jacobi_init

?jacobi delete

Releases allocated data.

Syntax

```
MKL_INT sjacobi_delete ( JACOBI_MATRIX_HANDLE t* handle);
```

```
MKL_INT djacobi_delete ( JACOBI_MATRIX_HANDLE t* handle);
```

Include Files

- mkl.h

Description

The `?jacobi_delete` routine releases all memory allocated for the handle. Only after calling this routine is it safe for the user to move or deallocate the memory referenced by `x` and `fiac`.

This routine flags memory as not used, but to actually release all memory you must call the support function `mkf free buffers`.

Input Parameters

<i>handle</i>	Type	JACOBI MATRIX HANDLE t.
---------------	------	-------------------------

Output Parameters

res Informs about the task completion.

res = TR_SUCCESS means the routine completed the task normally.

TR_SUCCESS is defined in the `mk1_rci.h` include file.

?jacobi

Computes the Jacobian matrix of the objective function using the central difference algorithm.

Syntax

```

MKL_INT sjacobi (USRFCNS fcn, const MKL_INT* n, const MKL_INT* m, float* fjac, float*
x, float* eps);

```

```

MKL_INT djacobi (USRFCND fcn, const MKL_INT* n, const MKL_INT* m, double* fjac, double*
x, double* eps);

```

Include Files

- mkl.h

Description

The `?jacobi` routine computes the Jacobian matrix for function `f_cn` using the central difference algorithm. This routine has a "Black-Box" interface, where you input the objective function via parameters. Your objective function must have a fixed interface.

See calling and usage examples in the `examples\solverc\source` folder of your Intel® oneAPI Math Kernel Library (oneMKL) directory. Specifically, see `ex_nlsqp_c.c` and `ex_nlsqp_bc.c.c`.

Input Parameters

fcn

User-supplied subroutine to evaluate the function that defines the least squares problem. Called as `fcn(m, n, x, f)` with the following parameters:

Parameter	Type	Description
Input Parameters		
<i>m</i>		Pointer to the length of <i>f</i> .
<i>n</i>		Pointer to the length of <i>x</i> .
<i>x</i>		Array of size <i>n</i> . Vector, at which the function is evaluated. The <code>fcn</code> function should not change this parameter.
Output Parameters		
<i>f</i>		Array of size <i>m</i> ; contains the function values at <i>x</i> .

You need to declare `fcn` as `extern` in the calling program.

n

Length of *X*.

m

Length of *F*.

x

Array of size *n*. Vector at which the function is evaluated.

eps

Precision of the Jacobian matrix calculation.

Output Parameters

fjac

Array of size *m* by *n*. Contains the Jacobian matrix of the function.

res

Indicates task completion status.

- *res* = `TR_SUCCESS` - the routine completed the task normally.
- *res* = `TR_INVALID_OPTION` - there was an error in the input parameters.
- *res* = `TR_OUT_OF_MEMORY` - there was a memory error.

`TR_SUCCESS`, `TR_INVALID_OPTION`, and `TR_OUT_OF_MEMORY` are defined in the `mkl_rci.h` include file.

See Also

[?jacobix](#)

?jacobix

Alternative interface for `?jacobi` function for passing additional data into the objective function.

Syntax

```
MKL_INT sjacobix (USRFCNXS fcn, const MKL_INT* n, const MKL_INT * m, float* fjac,
float* x, float* eps, void* user_data);
```

```
MKL_INT djacobix (USRFCNXD fcn, const MKL_INT* n, const MKL_INT * m, double* fjac,
double* x, double* eps, void* user_data);
```

Include Files

- `mkl.h`

Description

The `?jacobi` routine presents an alternative interface for the `?jacobi` function that enables you to pass additional data into the objective function `fcn`.

See calling and usage examples in the `examples\solverc\source` folder of your Intel® oneAPI Math Kernel Library (oneMKL) directory. Specifically, see `ex_nlsqp_c_x.c` and `ex_nlsqp_bc_c_x.c`.

Input Parameters

`fcn` User-supplied subroutine to evaluate the function that defines the least squares problem. Called as `fcn(m, n, x, f, user_data)` with the following parameters:

Parameter	Description
Input Parameters	
<code>m</code>	Pointer to the length of <code>f</code> .
<code>n</code>	Pointer to the length of <code>x</code> .
<code>x</code>	Array of size <code>n</code> . Vector, at which the function is evaluated. The <code>fcn</code> function should not change this parameter.
<code>user_data</code>	Pointer to your additional data, if any. Otherwise, a dummy argument.
Output Parameters	
<code>f</code>	Array of size <code>m</code> ; contains the function values at <code>x</code> .

You need to declare `fcn` as `extern` in the calling program.

<code>n</code>	Length of <code>X</code> .
<code>m</code>	Length of <code>F</code> .
<code>x</code>	Array of size <code>n</code> . Vector at which the function is evaluated.
<code>eps</code>	Precision of the Jacobian matrix calculation.
<code>user_data</code>	Pointer to your additional data. If there is no additional data, this is a dummy argument.

Output Parameters

`fjac` Array of size `m` by `n`). Contains the Jacobian matrix of the function.

`res` Indicates task completion status.

- `res = TR_SUCCESS` - the routine completed the task normally.
- `res = TR_INVALID_OPTION` - there was an error in the input parameters.
- `res = TR_OUT_OF_MEMORY` - there was a memory error.

TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.h include file.

See Also

?jacobi

Support Functions

Intel® oneAPI Math Kernel Library (oneMKL) support functions are subdivided into the following groups according to their purpose:

[Version Information](#)

[Threading Control](#)

[Error Handling](#)

[Character Equality Testing](#)

[Timing](#)

[Memory Management](#)

[Single Dynamic Library Control](#)

[Conditional Numerical Reproducibility Control](#)

[Miscellaneous](#)

The following table lists Intel® oneAPI Math Kernel Library (oneMKL) support functions.

oneMKL Support Functions

Function Name	Operation
Version Information	
<code>mkl_get_version</code>	Returns the Intel® oneAPI Math Kernel Library (oneMKL) version.
<code>mkl_get_version_string</code>	Returns the Intel® oneAPI Math Kernel Library (oneMKL) version in a character string.
Threading Control	
<code>mkl_set_num_threads</code>	Specifies the number of OpenMP* threads to use.
<code>mkl_domain_set_num_threads</code>	Specifies the number of OpenMP* threads for a particular function domain.
<code>mkl_set_num_threads_local</code>	Specifies the number of OpenMP* threads for all Intel® oneAPI Math Kernel Library (oneMKL) functions on the current execution thread.
<code>mkl_set_dynamic</code>	Enables Intel® oneAPI Math Kernel Library (oneMKL) to dynamically change the number of OpenMP* threads.
<code>mkl_get_max_threads</code>	Gets the number of OpenMP* threads targeted for parallelism.
<code>mkl_domain_get_max_threads</code>	Gets the number of OpenMP* threads targeted for parallelism for a particular function domain.
<code>mkl_get_dynamic</code>	Determines whether Intel® oneAPI Math Kernel Library (oneMKL) is enabled to dynamically change the number of OpenMP* threads.

Function Name	Operation
<code>mkl_set_num_stripes</code>	Specifies the number of partitions along the leading dimension of the output matrix for parallel ?gemm functions.
<code>mkl_get_num_stripes</code>	Gets the number of partitions along the leading dimension of the output matrix for parallel ?gemm functions.
Error Handling	
<code>xerbla</code>	Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.
<code>pxerbla</code>	Handles error conditions for the ScaLAPACK routines.
<code>LAPACKE_xerbla</code>	Error handling function called by the C interface to LAPACK functions.
<code>mkl_set_exit_handler</code>	Sets the custom handler of fatal errors.
Character Equality Testing	
<code>lsame</code>	Tests two characters for equality regardless of the case.
<code>lsamen</code>	Tests two character strings for equality regardless of the case.
Timing	
<code>second/dsecnd</code>	Returns elapsed time in seconds. Use to estimate real time between two calls to this function.
<code>mkl_get_cpu_clocks</code>	Returns elapsed CPU clocks.
<code>mkl_get_cpu_frequency</code>	Returns CPU frequency value in GHz.
<code>mkl_get_max_cpu_frequency</code>	Returns the maximum CPU frequency value in GHz.
<code>mkl_get_clocks_frequency</code>	Returns the frequency value in GHz based on constant-rate Time Stamp Counter.
Memory Management	
<code>mkl_free_buffers</code>	Frees unused memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.
<code>mkl_thread_free_buffers</code>	Frees unused memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator in the current thread.
<code>mkl_mem_stat</code>	Reports the status of the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.
<code>mkl_peak_mem_usage</code>	Reports the peak memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.
<code>mkl_disable_fast_mm</code>	Turns off the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator for Intel® oneAPI Math Kernel Library (oneMKL) functions to directly use the <code>systemmalloc/free</code> functions.
<code>mkl_malloc</code>	Allocates an aligned memory buffer.

Function Name	Operation
<code>mkl_calloc</code>	Allocates and initializes an aligned memory buffer.
<code>mkl_realloc</code>	Changes the size of memory buffer allocated by <code>mkl_malloc/mkl_calloc</code> .
<code>mkl_free</code>	Frees the aligned memory buffer allocated by <code>mkl_malloc/mkl_calloc</code> .
<code>mkl_set_memory_limit</code>	On Linux, sets the limit of memory that Intel® oneAPI Math Kernel Library (oneMKL) can allocate for a specified type of memory.
Single Dynamic Library (SDL) Control	
<code>mkl_set_interface_layer</code>	Sets the interface layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time.
<code>mkl_set_threading_layer</code>	Sets the threading layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time.
<code>mkl_set_xerbla</code>	Replaces the error handling routine. Use with the Single Dynamic Library .
<code>mkl_set_progress</code>	Replaces the progress information routine.
<code>mkl_set_pardiso_pivot</code>	Replaces the routine handling Intel® oneAPI Math Kernel Library (oneMKL) PARDISO pivots with a user-defined routine. Use with the Single Dynamic Library (SDL).
Conditional Numerical Reproducibility (CNR) Control	
<code>mkl_cbwr_set</code>	Configures the CNR mode of Intel® oneAPI Math Kernel Library (oneMKL).
<code>mkl_cbwr_get</code>	Returns the current CNR settings.
<code>mkl_cbwr_get_auto_branch</code>	Automatically detects the CNR code branch for your platform.
Miscellaneous	
<code>mkl_progress</code>	Provides progress information.
<code>mkl_enable_instructions</code>	
<code>mkl_set_env_mode</code>	Set up the mode that ignores environment settings specific to Intel® oneAPI Math Kernel Library (oneMKL).
<code>mkl_verbose</code>	Enable or disable Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode.
<code>mkl_verbose_output_file</code>	Write output in Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode to a file.
<code>mkl_set_mpi</code>	Sets the implementation of the message-passing interface to be used by Intel® oneAPI Math Kernel Library (oneMKL).
<code>mkl_finalize</code>	Terminates Intel® oneAPI Math Kernel Library (oneMKL) execution environment and frees resources allocated by the library.

Product and Performance Information

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

Notice revision #20201201

Version Information

Intel® oneAPI Math Kernel Library (oneMKL) provides two methods for extracting information about the library version number:

- extracting a version string using the `mkl_get_version_string` function
- using the `mkl_get_version` function to obtain an `MKLVersion` structure that contains the version information

A makefile is also provided to automatically build the examples and output summary files containing the version information for the current library.

mkl_get_version

Returns the Intel® oneAPI Math Kernel Library (oneMKL) version.

Syntax

```
void mkl_get_version( MKLVersion* pVersion );
```

Include Files

- `mkl.h`

Output Parameters

`pVersion` Pointer to the `MKLVersion` structure.

Description

The `mkl_get_version` function collects information about the active C version of the Intel® oneAPI Math Kernel Library (oneMKL) software and returns this information in a structure of `MKLVersion` type by the `pVersion` address. The `MKLVersion` structure type is defined in the `mkl_types.h` file. The following fields of the `MKLVersion` structure are available:

<code>MajorVersion</code>	is the major number of the current library version.
<code>MinorVersion</code>	is the minor number of the current library version.
<code>UpdateVersion</code>	is the update number of the current library version.
<code>ProductStatus</code>	is the status of the current library version. Possible variants are "Beta" or "Product".
<code>Build</code>	is the string that contains the build date and the internal build number.
<code>Platform</code>	is the string that contains the current architecture. Possible variants are "IA-32 architecture" or "Intel(R) 64 architecture".

Processor

is the processor optimization. Normally it is targeted for the processor installed on your system and based on the detection of the Intel® oneAPI Math Kernel Library (oneMKL) library that is optimal for the installed processor. In the Conditional Numerical Reproducibility (CNR) mode, the processor optimization matches the selected CNR branch.

Product and Performance Information

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

Notice revision #20201201

mkl_get_version Usage

```
#include <stdio.h>
#include <stdlib.h>
#include "mkl.h"

int main(void)
{
    MKLVersion Version;

    mkl_get_version(&Version);

    printf("Major version:      %d\n", Version.MajorVersion);
    printf("Minor version:      %d\n", Version.MinorVersion);
    printf("Update version:      %d\n", Version.UpdateVersion);
    printf("Product status:      %s\n", Version.ProductStatus);
    printf("Build:                %s\n", Version.Build);
    printf("Platform:            %s\n", Version.Platform);
    printf("Processor optimization: %s\n", Version.Processor);
    printf("=====\n");
    printf("\n");

    return 0;
}
```

Output:

Major Version	11
Minor Version	0
Update Version	2
Product status	Product
Build	20121113
Platform	Intel(R) 64 architecture
Processor optimization	Intel(R) Core(TM) i7 Processor

See Also

[Conditional Numerical Reproducibility Control](#)

mkl_get_version_string

Returns the Intel® oneAPI Math Kernel Library (oneMKL) version in a character string.

Syntax

```
void mkl_get_version_string (char* buf, int len);
```

Include Files

- mkl.h

Output Parameters

Name	Type	Description
<i>buf</i>	char*	Source string
<i>len</i>	int	Length of the source string

Description

The function returns a string that contains the Intel® oneAPI Math Kernel Library (oneMKL) version.

For usage details, see the code example below:

Example

```
#include <stdio.h>
#include "mkl.h"

int main(void)
{
    int len=198;
    char buf[198];
    mkl_get_version_string(buf, len);
    printf("%s\n",buf);
    printf("\n");
    return 0;
}
```

Threading Control

Intel® oneAPI Math Kernel Library (oneMKL) provides functions for OpenMP* threading control, discussed in this section.

Important

If Intel® oneAPI Math Kernel Library (oneMKL) operates within the Intel® Threading Building Blocks (Intel® TBB) execution environment, the environment variables for OpenMP* threading control, such as `asomp_num_threads`, and Intel® oneAPI Math Kernel Library (oneMKL) functions discussed in this section have no effect. If the Intel TBB threading technology is used, control the number of threads through the Intel TBB application programming interface. Read the documentation for the `tbb::task_scheduler_init` class at <https://www.threadingbuildingblocks.org/docs/doxygen/a00150.html> to find out how to specify the number of Intel TBB threads.

If Intel® oneAPI Math Kernel Library (oneMKL) operates within an OpenMP* execution environment, you can control the number of threads for Intel® oneAPI Math Kernel Library (oneMKL) using OpenMP* runtime library routines and environment variables (see the OpenMP* specification for details). Additionally Intel® oneAPI

Math Kernel Library (oneMKL) provides *optional* threading control functions and environment variables that enable you to specify the number of threads for Intel® oneAPI Math Kernel Library (oneMKL) and to control dynamic adjustment of the number of threads *independently* of the OpenMP* settings. The settings made with the Intel® oneAPI Math Kernel Library (oneMKL) threading control functions and environment variables do not affect OpenMP* settings but take precedence over them.

If functions are used, Intel® oneAPI Math Kernel Library (oneMKL) environment variables may control Intel® oneAPI Math Kernel Library (oneMKL) threading. For details of those environment variables, see the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

You can specify the number of threads for Intel® oneAPI Math Kernel Library (oneMKL) function domains with the [mkl_set_num_threads](#) or [mkl_domain_set_num_threads](#) function. While [mkl_set_num_threads](#) specifies the number of threads for the entire Intel® oneAPI Math Kernel Library (oneMKL), [mkl_domain_set_num_threads](#) does it for a specific function domain. The following table lists the function domains that support independent threading control. The table also provides named constants to pass to threading control functions as a parameter that specifies the function domain.

oneMKL Function Domains

Function Domain	Named Constant
Basic Linear Algebra Subroutines (BLAS)	MKL_DOMAIN_BLAS
Fast Fourier Transform (FFT) functions, except Cluster FFT functions	MKL_DOMAIN_FFT
Vector Math (VM) functions	MKL_DOMAIN_VML
Parallel Direct Solver (PARDISO) functions	MKL_DOMAIN_PARDISO
All Intel® oneAPI Math Kernel Library (oneMKL) functions except the functions from the domains where the number of threads is set explicitly.	MKL_DOMAIN_ALL

Warning

Do not increase the number of OpenMP threads used for `cluster_sparse_solver` between the first call and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.

Both [mkl_set_num_threads](#) and [mkl_domain_set_num_threads](#) functions set the number of threads for all subsequent calls to Intel® oneAPI Math Kernel Library (oneMKL) from all applications threads. Use the [mkl_set_num_threads_local](#) function to specify different numbers of threads for Intel® oneAPI Math Kernel Library (oneMKL) on different execution threads of your application. The thread-local settings take precedence over the global settings. However, the thread-local settings may have undesirable side effects (see the description of the [mkl_set_num_threads_local](#) function for details).

By default, Intel® oneAPI Math Kernel Library (oneMKL) can adjust the specified number of threads dynamically. For example, Intel® oneAPI Math Kernel Library (oneMKL) may use fewer threads if the size of the computation is not big enough or not create parallel regions when running within an OpenMP* parallel region. Although Intel® oneAPI Math Kernel Library (oneMKL) may actually use a different number of threads from the number specified, the library does not create parallel regions with more threads than specified. If dynamic adjustment of the number of threads is disabled, Intel® oneAPI Math Kernel Library (oneMKL) attempts to use the specified number of threads in internal parallel regions (for more information, see the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*). Use the [mkl_set_dynamic](#) function to control dynamic adjustment of the number of threads.

[mkl_set_num_threads](#)

Specifies the number of OpenMP threads to use.*

Syntax

```
void mkl_set_num_threads( int nt );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>nt</i>	int	<i>nt</i> > 0 - The number of threads suggested by the user. <i>nt</i> ≤ 0 - Invalid value, which is ignored.

Description

This function enables you to specify how many OpenMP threads Intel® oneAPI Math Kernel Library (oneMKL) should use for internal parallel regions. If this number is not set (default), Intel® oneAPI Math Kernel Library (oneMKL) functions use the default number of threads for the OpenMP run-time library. The specified number of threads applies:

- To all Intel® oneAPI Math Kernel Library (oneMKL) functions except the functions from the domains where the number of threads is set with [mkl_domain_set_num_threads](#)
- To all execution threads except the threads where the number of threads is set with [mkl_set_num_threads_local](#)

The number specified is a hint, and Intel® oneAPI Math Kernel Library (oneMKL) may actually use a smaller number.

NOTE

This function takes precedence over the `MKL_NUM_THREADS` environment variable.

Example

```
#include "mkl.h"
...
mkl_set_num_threads(4);
my_compute_using_mkl();    // Intel MKL uses up to 4 OpenMP threads
```

mkl_domain_set_num_threads

Specifies the number of OpenMP threads for a particular function domain.*

Syntax

```
int mkl_domain_set_num_threads (int nt, int domain);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>nt</i>	int	<i>nt</i> > 0 - The number of threads suggested by the user.

Name	Type	Description
		<i>nt</i> = 0 - The default number of threads for the OpenMP run-time library.
		<i>nt</i> < 0 - Invalid value, which is ignored.
<i>domain</i>	int	The named constant that defines the targeted domain.

Description

This function specifies how many OpenMP threads a particular function domain of Intel® oneAPI Math Kernel Library (oneMKL) should use. If this number is not set (default) or if it is set to zero in a call to this function, Intel® oneAPI Math Kernel Library (oneMKL) uses the default number of threads for the OpenMP run-time library. The number of threads specified applies to the specified function domain on all execution threads except the threads where the number of threads is set with [mkl_set_num_threads_local](#). For a list of supported values of the *domain* argument, see [Table "Intel MKL Function Domains"](#).

The number of threads specified is only a hint, and Intel® oneAPI Math Kernel Library (oneMKL) may actually use a smaller number.

NOTE

This function takes precedence over the `MKL_DOMAIN_NUM_THREADS` environment variable.

Return Values

Name	Type	Description
<i>ierr</i>	int	1 - Indicates no error, execution is successful.
		0 - Indicates a failure, possibly because of invalid input parameters.

Example

```
#include "mkl.h"
...
mkl_domain_set_num_threads(4, MKL_DOMAIN_BLAS);
my_compute_using_mkl_blas();    // Intel MKL BLAS functions use up to 4 threads
my_compute_using_mkl_dft();     // Intel MKL FFT functions use the default number of threads
```

mkl_set_num_threads_local

Specifies the number of OpenMP threads for all Intel® oneAPI Math Kernel Library (oneMKL) functions on the current execution thread.*

Syntax

```
int mkl_set_num_threads_local( int nt );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>nt</i>	int	<p><i>nt</i> > 0 - The number of threads for Intel® oneAPI Math Kernel Library (oneMKL) functions to use on the current execution thread.</p> <p><i>nt</i> = 0 - A request to reset the thread-local number of threads and use the global number.</p>

Description

This function sets the number of OpenMP threads that Intel® oneAPI Math Kernel Library (oneMKL) functions should request for parallel computation. The number of threads is thread-local, which means that it only affects the current execution thread of the application. If the thread-local number is not set or if this number is set to zero in a call to this function, Intel® oneAPI Math Kernel Library (oneMKL) functions use the global number of threads. You can set the global number of threads using the [mkl_set_num_threads](#) or [mkl_domain_set_num_threads](#) function.

The thread-local number of threads takes precedence over the global number: if the thread-local number is non-zero, changes to the global number of threads have no effect on the current thread.

Caution

If your application is threaded with OpenMP* and parallelization of Intel® oneAPI Math Kernel Library (oneMKL) is based on nested OpenMP parallelism, different OpenMP parallel regions reuse OpenMP threads. Therefore a thread-local setting in one OpenMP parallel region may continue to affect not only the master thread after the parallel region ends, but also subsequent parallel regions. To avoid performance implications of this side effect, reset the thread-local number of threads before leaving the OpenMP parallel region (see [Examples](#) for how to do it).

Return Values

Name	Type	Description
<i>save_nt</i>	int	The value of the thread-local number of threads that was used before this function call. Zero means that the global number of threads was used.

Examples

This example shows how to avoid the side effect of a thread-local number of threads by reverting to the global setting:

```
#include "omp.h"
#include "mkl.h"
...
mkl_set_num_threads(16);
my_compute_using_mkl();           // Intel MKL functions use up to 16 threads
#pragma omp parallel num_threads(2)
{
    if (0 == omp_get_thread_num())
        mkl_set_num_threads_local(4);
    else
        mkl_set_num_threads_local(12);
}
```

```

    my_compute_using_mkl();          // Intel MKL functions use up to 4 threads on thread 0
//    and up to 12 threads on thread 1
}
my_compute_using_mkl();             // Intel MKL functions use up to 4 threads (!)
mkl_set_num_threads_local( 0 );    // make master thread use global setting
my_compute_using_mkl();             // Intel MKL functions use up to 16 threads

```

This example shows how to avoid the side effect of a thread-local number of threads by saving and restoring the existing setting:

```

#include "mkl.h"
void my_compute( int nt )
{
    int save = mkl_set_num_threads_local( nt ); // save the Intel® oneAPI Math Kernel Library
(oneMKL) number of threads
    my_compute_using_mkl();          // Intel MKL functions use up to nt threads on this thread
    mkl_set_num_threads_local( save ); // restore the Intel® oneAPI Math Kernel Library (oneMKL)
number of threads
}

```

mkl_set_dynamic

Enables Intel® oneAPI Math Kernel Library (oneMKL) to dynamically change the number of OpenMP* threads.

Syntax

```
void mkl_set_dynamic (int flag);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>flag</i>	int	<p><i>flag</i> = 0 - Requests disabling dynamic adjustment of the number of threads.</p> <p><i>flag</i> ≠ 0 - Requests enabling dynamic adjustment of the number of threads.</p>

Description

This function indicates whether Intel® oneAPI Math Kernel Library (oneMKL) can dynamically change the number of OpenMP threads or should avoid doing this. The setting applies to all Intel® oneAPI Math Kernel Library (oneMKL) functions on all execution threads. This function takes precedence over the `MKL_DYNAMIC` environment variable.

Dynamic adjustment of the number of threads is enabled by default. Specifically, Intel® oneAPI Math Kernel Library (oneMKL) may use fewer threads in parallel regions than the number returned by the [mkl_get_max_threads](#) function. Disabling dynamic adjustment of the number of threads does not ensure that Intel® oneAPI Math Kernel Library (oneMKL) actually uses the specified number of threads, although the library attempts to use that number.

Tip

If you call Intel® oneAPI Math Kernel Library (oneMKL) from within an OpenMP parallel region and want to create internal parallel regions, either disable dynamic adjustment of the number of threads or set the thread-local number of threads ([seemkl_set_num_threads_local](#) for how to do it).

Example

```
#include "mkl.h"
...
mkl_set_num_threads( 8 );
#pragma omp parallel
{
    my_compute_with_mkl();    // Intel MKL uses 1 thread, being called from OpenMP parallel
region
    mkl_set_dynamic( 0 );    // disable adjustment of the number of threads
    my_compute_with_mkl();    // Intel MKL uses 8 threads
}
```

mkl_get_max_threads

Gets the number of OpenMP threads targeted for parallelism.*

Syntax

```
int mkl_get_max_threads (void);
```

Include Files

- mkl.h

Description

This function returns the number of OpenMP threads available for Intel® oneAPI Math Kernel Library (oneMKL) to use in internal parallel regions.

Return Values

Name	Type	Description
<i>nt</i>	int	The maximum number of threads for Intel® oneAPI Math Kernel Library (oneMKL) functions to use in internal parallel regions.

Example

```
#include "mkl.h"
...
if (1 == mkl_get_max_threads()) puts("Intel MKL does not employ threading");
```

See Also

[mkl_set_dynamic](#)

[mkl_get_dynamic](#)

mkl_domain_get_max_threads

Gets the number of OpenMP threads targeted for parallelism for a particular function domain.*

Syntax

```
int mkl_domain_get_max_threads (int domain);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>domain</i>	int	The named constant that defines the targeted domain.

Description

Computational functions of the Intel® oneAPI Math Kernel Library (oneMKL) function domain defined by the *domain* parameter use the value returned by this function as a limit of the number of OpenMP threads they should request for parallel computations. The `mkl_domain_get_max_threads` function returns the thread-local number of threads or, if that value is zero or not set, the global number of threads. To determine this number, the function inspects the environment settings and return values of the function calls below in the order they are listed until it finds a non-zero value:

- A call to `mkl_set_num_threads_local`
- The last of the calls to `mkl_set_num_threads` or `mkl_domain_set_num_threads(..., MKL_DOMAIN_ALL)`
- A call to `mkl_domain_set_num_threads(..., domain)`
- The `MKL_DOMAIN_NUM_THREADS` environment variable with the `MKL_DOMAIN_ALL` tag
- The `MKL_DOMAIN_NUM_THREADS` environment variable (with the specific domain tag)
- The `MKL_NUM_THREADS` environment variable
- A call to `omp_set_num_threads`
- The `OMP_NUM_THREADS` environment variable

Actual number of threads used by the Intel® oneAPI Math Kernel Library (oneMKL) computational functions may vary depending on the problem size and on whether dynamic adjustment of the number of threads is enabled (see the description of `mkl_set_dynamic`). For a list of supported values of the *domain* argument, see [Table "Intel MKL Function Domains"](#).

Return Values

Name	Type	Description
<i>nt</i>	int	The maximum number of threads for Intel® oneAPI Math Kernel Library (oneMKL) functions from a given domain to use in internal parallel regions. If an invalid value of <i>domain</i> is supplied, the function returns the number of threads for <code>MKL_DOMAIN_ALL</code>

Example

```
#include "mkl.h"
...
if (1 < mkl_domain_get_max_threads(MKL_DOMAIN_BLAS))
puts("Intel MKL BLAS functions employ threading");
```

mkl_get_dynamic

Determines whether Intel® oneAPI Math Kernel Library (oneMKL) is enabled to dynamically change the number of OpenMP threads.*

Syntax

```
int mkl_get_dynamic(void);
```

Include Files

- `mkl.h`

Description

This function returns the status of dynamic adjustment of the number of OpenMP* threads. To determine this status, the function inspects the return value of the following function call and if it is undefined, inspects the environment setting below:

- A call to [mkl_set_dynamic](#)
- The `MKL_DYNAMIC` environment variable

NOTE

Dynamic adjustment of the number of threads is enabled by default.

The dynamic adjustment works as follows. Suppose that the [mkl_get_max_threads](#) function returns the number of threads equal to N . If dynamic adjustment is enabled, Intel® oneAPI Math Kernel Library (oneMKL) may request up to N threads, depending on the size of the problem. If dynamic adjustment is disabled, Intel® oneAPI Math Kernel Library (oneMKL) requests exactly N threads for internal parallel regions (provided it uses a threaded algorithm with at least N computations that can be done in parallel). However, the OpenMP* run-time library may be configured to supply fewer threads than Intel® oneAPI Math Kernel Library (oneMKL) requests, depending on the OpenMP* setting of dynamic adjustment.

Return Values

Name	Type	Description
<code>ret</code>	<code>int</code>	0 - Dynamic adjustment of the number of threads is disabled. 1 - Dynamic adjustment of the number of threads is enabled.

Example

```
#include "mkl.h"
...
int nt = mkl_get_max_threads();
if (1 == mkl_get_dynamic())
printf("Intel MKL may use less than %i threads for a large problem", nt);
else
printf("Intel MKL should use %i threads for a large problem", nt);
```

mkl_set_num_stripes

Specifies the number of partitions along the leading dimension of the output matrix for parallel ?gemm functions.

Syntax

```
void mkl_set_num_stripes( int ns );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
ns	int	<p>ns > 0 - Specifies the number of partitions to use.</p> <p>ns = 0 - Instructs Intel® oneAPI Math Kernel Library (oneMKL) to use the default partitioning algorithm.</p> <p>ns < 0 - Invalid value; ignored.</p>

Description

This function enables you to specify the number of stripes, or partitions along the leading dimension of the output matrix, for parallel ?gemm functions. If this number is not set (default) or if it is set to zero, Intel® oneAPI Math Kernel Library (oneMKL) ?gemm functions use the default partitioning algorithm. The specified number of partitions only applies to ?gemm functions.

The number specified is a hint, and Intel® oneAPI Math Kernel Library (oneMKL) may actually use a smaller number.

NOTE

This function takes precedence over the MKL_NUM_STRIPES environment variable.

Example

```
#include "mkl.h"
...
mkl_set_num_stripes(4);
dgemm(...);    // Intel MKL uses up to 4 stripes for dgemm
```

See Also

[mkl_get_num_stripes](#)

mkl_get_num_stripes

Gets the number of partitions along the leading dimension of the output matrix for parallel ?gemm functions.

Syntax

```
int mkl_get_num_stripes(void);
```

Include Files

- `mkl.h`

Description

This function returns the number of stripes, that is, partitions along the leading dimension of the output matrix, for parallel `?gemm` functions. The number of partitions only applies to `?gemm` functions.

The number returned is a hint, and Intel® oneAPI Math Kernel Library (oneMKL) may actually use a smaller number.

Return Values

Name	Type	Description
<code>ns</code>	<code>int</code>	The number of stripes for Intel® oneAPI Math Kernel Library (oneMKL) <code>?gemm</code> functions to use.

Example

```
#include "mkl.h"
...
int ns = mkl_get_num_stripes();
if (ns > 0) printf("Intel MKL uses %d number of stripes\n", ns);
```

See Also

[mkl_set_num_stripes](#)

Error Handling

Error Handling for Linear Algebra Routines

xerbla

Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.

Syntax

```
void xerbla( const char * sname, const int* info, const int len );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>sname</code>	<code>const char*</code>	The name of the routine that called <code>xerbla</code>
<code>info</code>	<code>const int*</code>	The position of the invalid parameter in the parameter list of the calling function or an error code
<code>len</code>	<code>const int</code>	Length of the source string

Description

The `xerbla` function is an error handler for Intel® oneAPI Math Kernel Library (oneMKL) BLAS, LAPACK, Vector Math, and Vector Statistics functions. These functions call `xerbla` if an issue is encountered on entry or during the function execution.

`xerbla` operates as follows:

1. Prints a message that depends on the value of the `info` parameter as explained in the following table.

NOTE

A specific message can differ from the listed messages in numeric values and/or function names.

2. Returns to the calling application.

Error Messages Printed by `xerbla`

Value of <code>info</code>	Error Message
1001	Intel MKL ERROR: Incompatible optional parameters on entry to DGEMM.
1000 or 1089	Intel MKL INTERNAL ERROR: Insufficient workspace available in function CGELSD.
< 0	Intel MKL INTERNAL ERROR: Condition 1 detected in function DLASD8.
Other	The position of the invalid parameter in the parameter list of the calling function.

Note that `xerbla` is an internal function. You can change or disable printing of an error message by providing your own `xerbla` function. The following examples illustrate usage of `xerbla`.

Example

```
void xerbla(char* sname, int* info, int len){
// sname - name of the function that called xerbla
// info - position of the invalid parameter in the parameter list
// len - length of the name in bytes
printf("\nXERBLA is called :%s: %d\n",sname,*info);
}
```

See Also

[`mkl_set_xerbla`](#)

`pxerbla`

Error handling routine called by ScaLAPACK routines.

Syntax

```
void pxerbla (MKL_INT* ictxt, char* sname, MKL_INT* info, MKL_INT sname_len);
```

Include Files

- `mkl_scalapack.h`

Input Parameters

`ictxt` (local)

	MKL_INT*	
		The BLACS context handle, indicating the global context of the operation. The context itself is global.
<i>sname</i>	(global) char*	
		The name of the routine that called <code>pserbla</code> .
<i>info</i>	(global) MKL_INT*	
		The position of the invalid parameter in the parameter list of the calling routine.
<i>sname_len</i>	(global) MKL_INT	
		The length of the calling routine name.

Description

This routine is an error handler for the *ScaLAPACK* routines. It is called if an input parameter has an invalid value. A message is printed and program execution continues. For *ScaLAPACK* driver and computational routines, a `RETURN` statement is issued following the call to `pserbla`.

Control returns to the higher-level calling routine, and you can determine how the program should proceed. However, in the specialized low-level *ScaLAPACK* routines (auxiliary routines that are Level 2 equivalents of computational routines), the call to `pserbla()` is immediately followed by a call to `BLACS_ABORT()` to terminate program execution since recovery from an error at this level in the computation is not possible.

It is always good practice to check for a non-zero value of *info* on return from a *ScaLAPACK* routine. Installers may consider modifying this routine in order to call system-specific exception-handling facilities.

LAPACKE_xerbla

Error handling function called by the C interface to LAPACK functions.

Syntax

```
void LAPACKE_xerbla( const char * name, lapack_int info );
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>name</i>	const char*	The name of the routine that called <code>LAPACKE_xerbla</code>
<i>info</i>	lapack_int	The position of the invalid parameter in the parameter list of the calling function or an error code

Description

The `LAPACKE_xerbla` function is an error handler for Intel® oneAPI Math Kernel Library (oneMKL) LAPACKE functions (the C interface to LAPACK functionality). If a LAPACKE function encounters an issue on entry or during the function execution, it calls `LAPACKE_xerbla` to print an error message and return an error code.

NOTE

The `LAPACKE_xerbla` routine does not replace the `xerbla` routine. For instance, if an issue occurs when a LAPACK function is called by a LAPACKE function, the LAPACK function calls `xerbla`.

Error Messages Printed by LAPACKE_xerbla

Value of <i>info</i>	Example Error Message
<code>LAPACK_WORK_MEMORY_ERROR</code>	Not enough memory to allocate work array in <code>LAPACKE_dgees</code>
<code>LAPACK_TRANSPOSE_MEMORY_ERROR</code>	Not enough memory to transpose matrix in <code>LAPACKE_dgetrf_work</code>
<code>< 0</code>	Wrong parameter 1 in <code>LAPACKE_dgetrf</code>

NOTE

`LAPACKE_xerbla` is an internal function. You can change or disable printing of an error message by providing your own `LAPACKE_xerbla` function. Intel® oneAPI Math Kernel Library (oneMKL) does not provide functionality for dynamic replacement of `LAPACKE_xerbla`.

See Also

[xerbla](#)

Handling Fatal Errors

A fatal error is a circumstance under which Intel® oneAPI Math Kernel Library (oneMKL) cannot continue the computation. For example, a fatal error occurs when Intel® oneAPI Math Kernel Library (oneMKL) cannot load a dynamic library or confronts an unsupported CPU type. In case of a fatal error, the default Intel® oneAPI Math Kernel Library (oneMKL) behavior is to print an explanatory message to the console and call an internal function that terminates the application with a call to the `system_exit()` function. Intel® oneAPI Math Kernel Library (oneMKL) enables you to override this behavior by setting a custom handler of fatal errors. The custom error handler can be configured to throw a C++ exception, set a global variable indicating the failure, or otherwise handle cannot-continue situations. It is not necessary for the custom error handler to call the `system_exit()` function. Once execution of the error handler completes, a call to Intel® oneAPI Math Kernel Library (oneMKL) returns to the calling program without performing any computations and leaves no memory allocated by Intel® oneAPI Math Kernel Library (oneMKL) and no thread synchronization pending on return.

To specify a custom fatal error handler, call the [mkl_set_exit_handler](#) function.

mkl_set_exit_handler

Sets the custom handler of fatal errors.

Syntax

```
int mkl_set_exit_handler (MKLExitHandler myexit);
```


Include Files

- mkl.h

Input Parameters

Name	Prototype	Description
<i>myexit</i>	<code>void (*myexit)(int why);</code>	The error handler to set.

Description

This function sets the custom handler of fatal errors. If the input parameter is *NULL*, the system `exit()` function is set.

The following example shows how to use a custom handler of fatal errors in your C++ application:

```
#include "mkl.h"
void my_exit(int why){
    throw my_exception();
}
int ComputationFunction()
{
    mkl_set_exit_handler( my_exit );
    try {
        compute_using_mkl();
    }
    catch (const my_exception& e) {
        handle_exception();
    }
}
```

Character Equality Testing

lsame

Tests two characters for equality regardless of the case.

Syntax

```
int lsame( const char* ca, const char* cb, int lca, int lcb );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>ca, cb</i>	<code>const char*</code>	Pointers to the single characters to be compared
<i>lca, lcb</i>	<code>int</code>	Lengths of the input character strings, equal to one.

Description

This logical function checks whether two characters are equal regardless of the case.

Return Values

Name	Type	Description
<i>val</i>	int	Result of the comparison: <ul style="list-style-type: none"> a non-zero value if <i>ca</i> is the same letter as <i>cb</i>, maybe except for the case. zero if <i>ca</i> and <i>cb</i> are different letters for whatever cases.

lsamen

Tests two character strings for equality regardless of the case.

Syntax

```
MKL_INT lsamen( const MKL_INT* n, const char* ca, const char* cb );
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	const MKL_INT*	Pointer to the number of characters in <i>ca</i> and <i>cb</i> to be compared.
<i>ca, cb</i>	const char*	Character strings of length at least <i>n</i> to be compared. Only the first <i>n</i> characters of each string will be accessed.

Description

This logical function tests whether the first *n* letters of one string are the same as the first *n* letters of the other string, regardless of the case.

Return Values

Name	Type	Description
<i>val</i>	MKL_INT	Result of the comparison: <ul style="list-style-type: none"> a non-zero value if the first <i>n</i> letters in <i>ca</i> and <i>cb</i> character strings are equal, maybe except for the case, or if the length of character string <i>ca</i> or <i>cb</i> is less than <i>n</i>. zero if the first <i>n</i> letters in <i>ca</i> and <i>cb</i> character strings are different for whatever cases.

Timing

second/dsecnd

Returns elapsed time in seconds. Use to estimate real time between two calls to this function.

Syntax

```
float second( void );  
double dsecnd( void );
```

Include Files

- mkl.h

Description

The `second/dsecnd` function returns time in seconds to be used to estimate real time between two calls to the function. The difference between these functions is in the precision of the floating-point type of the result: while `second` returns the single-precision type, `dsecnd` returns the double-precision type.

Use these functions to measure durations. To do this, call each of these functions twice. For example, to measure performance of a routine, call the appropriate function directly before a call to the routine to be measured, and then after the call of the routine. The difference between the returned values shows real time spent in the routine.

Initializations may take some time when the `second/dsecnd` function runs for the first time. To eliminate the effect of this extra time on your measurements, make the first call to `second/dsecnd` in advance.

Do not use `second` to measure short time intervals because the single-precision format is not capable of holding sufficient timer precision.

Return Values

Name	Type	Description
<code>val</code>	<code>float</code> for <code>second</code> <code>double</code> for <code>dsecnd</code>	Elapsed real time in seconds

mkl_get_cpu_clocks

Returns elapsed CPU clocks.

Syntax

```
void mkl_get_cpu_clocks (unsigned MKL_INT64 *clocks);
```

Include Files

- mkl.h

Output Parameters

Name	Type	Description
<code>clocks</code>	<code>unsigned MKL_INT64</code>	Elapsed CPU clocks

Description

The `mkl_get_cpu_clocks` function returns the elapsed CPU clocks.

This may be useful when timing short intervals with high resolution. The `mkl_get_cpu_clocks` function is also applied in pairs like `second/dsecnd`. Note that out-of-order code execution on IA-32 or Intel® 64 architecture processors may disturb the exact elapsed CPU clocks value a little bit, which may be important while measuring extremely short time intervals.

Product and Performance Information

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

Notice revision #20201201

mkl_get_cpu_frequency

Returns the current CPU frequency value in GHz.

Syntax

```
double mkl_get_cpu_frequency(void);
```

Include Files

- `mkl.h`

Description

The function `mkl_get_cpu_frequency` returns the current CPU frequency in GHz.

NOTE

The returned value may vary from run to run if power management or Intel® Turbo Boost Technology is enabled.

Return Values

Name	Type	Description
<i>freq</i>	double	Current CPU frequency value in GHz

mkl_get_max_cpu_frequency

Returns the maximum CPU frequency value in GHz.

Syntax

```
double mkl_get_max_cpu_frequency(void);
```

Include Files

- `mkl.h`

Description

The function `mkl_get_max_cpu_frequency` returns the maximum CPU frequency in GHz.

Return Values

Name	Type	Description
<i>freq</i>	double	Maximum CPU frequency value in GHz

mkl_get_clocks_frequency

Returns the frequency value in GHz based on constant-rate Time Stamp Counter.

Syntax

```
double mkl_get_clocks_frequency (void);
```

Include Files

- `mkl.h`

Description

The function `mkl_get_clocks_frequency` returns the CPU frequency value (in GHz) based on constant-rate Time Stamp Counter (TSC). Use of the constant-rate TSC ensures that each clock tick is constant even if the CPU frequency changes. Therefore, the returned frequency is constant.

NOTE

Obtaining the frequency may take some time when `mkl_get_clocks_frequency` is called for the first time. The same holds for functions `second/dsecnd`, which call `mkl_get_clocks_frequency`.

Return Values

Name	Type	Description
<code>freq</code>	<code>double</code>	Frequency value in GHz

See Also

[second/dsecnd](#)

Memory Management

This section describes the Intel® oneAPI Math Kernel Library (oneMKL) memory functions. See the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for more memory usage information.

mkl_free_buffers

Frees unused memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) on the Host.

Syntax

```
void mkl_free_buffers (void);
```

Include Files

- `mkl.h`

Description

To improve performance of Intel® oneAPI Math Kernel Library (oneMKL) on CPU, the Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. Intel® oneAPI Math Kernel Library (oneMKL) also allocates temporary buffers on the host memory to improve performance of GPU kernels. The `mkl_free_buffers` function frees both types of memory.

See the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details.

You should call `mkl_free_buffers` after the last call to Intel® oneAPI Math Kernel Library (oneMKL) functions. In large applications, if you suspect that the memory may get insufficient, you may call this function earlier, but anticipate a drop in performance that may occur due to reallocation of buffers for subsequent calls to Intel® oneAPI Math Kernel Library (oneMKL) functions.

Product and Performance Information

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

Notice revision #20201201

Usage of `mkl_free_buffers` with FFT Functions

```
DFTI_DESCRIPTOR_HANDLE hand1;
DFTI_DESCRIPTOR_HANDLE hand2;
void mkl_free_buffers(void);
. . . . .
/* Using Intel MKL FFT */
Status = DftiCreateDescriptor(&hand1, DFTI_SINGLE, DFTI_COMPLEX, dim, m1);
Status = DftiCommitDescriptor(hand1);
Status = DftiComputeForward(hand1, s_array1);
. . . . .
Status = DftiCreateDescriptor(&hand2, DFTI_SINGLE, DFTI_COMPLEX, dim, m2);
Status = DftiCommitDescriptor(hand2);
. . . . .
Status = DftiFreeDescriptor(&hand1);
. . . . .
Status = DftiComputeBackward(hand2, s_array2));
Status = DftiFreeDescriptor(&hand2);
/* Here you finish using Intel MKL FFT */
/* Memory leak will be triggered by any memory control tool */
/* Use mkl_free_buffers() to avoid memory leaking */
mkl_free_buffers();
```

`mkl_thread_free_buffers`

Frees unused memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator in the current thread.

Syntax

```
void mkl_thread_free_buffers (void);
```

Include Files

- `mkl.h`

Description

To improve performance of Intel® oneAPI Math Kernel Library (oneMKL), the Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The `mkl_thread_free_buffers` function frees unused memory allocated by the Memory Allocator in the current thread only.

You should call `mkl_thread_free_buffers` after the last call to Intel® oneAPI Math Kernel Library (oneMKL) functions in the current thread. In large applications, if you suspect that the memory may get insufficient, you may call this function earlier, but anticipate a drop in performance that may occur due to reallocation of buffers for subsequent calls to Intel® oneAPI Math Kernel Library (oneMKL) functions.

See Also

[mkl_free_buffers](#)

mkl_disable_fast_mm

Turns off the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator for Intel® oneAPI Math Kernel Library (oneMKL) functions to directly use the `systemmalloc/free` functions.

Syntax

```
int mkl_disable_fast_mm (void);
```

Include Files

- `mkl.h`

Description

The `mkl_disable_fast_mm` function turns the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator off for Intel® oneAPI Math Kernel Library (oneMKL) functions to directly use the `systemmalloc/free` functions. Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The Memory Allocator is turned on by default for better performance. To turn it off, you can use the `mkl_disable_fast_mm` function or the `MKL_DISABLE_FAST_MM` environment variable (See the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details.) Call `mkl_disable_fast_mm` before calling any Intel® oneAPI Math Kernel Library (oneMKL) functions that require allocation of memory buffers.

NOTE

Turning the Memory Allocator off negatively impacts performance of some Intel® oneAPI Math Kernel Library (oneMKL) routines, especially for small problem sizes.

Return Values

Name	Type	Description
<code>mm</code>	<code>int</code>	1 - The Memory Allocator is successfully turned off. 0 - Turning the Memory Allocator off failed.

mkl_mem_stat

Reports the status of the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.

Syntax

```
MKL_INT64 mkl_mem_stat (int* AllocatedBuffers);
```

Include Files

- `mkl.h`

Output Parameters

Name	Type	Description
<i>AllocatedBuffers</i>	int	The number of buffers allocated by Intel® oneAPI Math Kernel Library (oneMKL).

Description

The function returns the number of buffers allocated by Intel® oneAPI Math Kernel Library (oneMKL) and the amount of memory in these buffers. Intel® oneAPI Math Kernel Library (oneMKL) can allocate the memory buffers internally or in a call to [mkl_malloc/mkl_calloc](#). If no buffers are allocated at the moment, the `mkl_mem_stat` function returns 0. Call `mkl_mem_stat` to check the Intel® oneAPI Math Kernel Library (oneMKL) memory status.

NOTE

If you free all the memory allocated in calls to `mkl_malloc` or `mkl_calloc` and then call [mkl_free_buffers](#), a subsequent call to `mkl_mem_stat` normally returns 0.

Return Values

Name	Type	Description
<i>AllocatedBytes</i>	MKL_INT64	The amount of allocated memory (in bytes).

See Also

[Usage Example for the Memory Functions](#)

[mkl_peak_mem_usage](#)

Reports the peak memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.

Syntax

```
MKL_INT64 mkl_peak_mem_usage (int mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>mode</i>	int	Requested mode of the function's operation. Possible values: <ul style="list-style-type: none"> • <code>MKL_PEAK_MEM_ENABLE</code> - start gathering the peak memory data • <code>MKL_PEAK_MEM_DISABLE</code> - stop gathering the peak memory data • <code>MKL_PEAK_MEM</code> - return the peak memory

Name	Type	Description
		<ul style="list-style-type: none"> MKL_PEAK_MEM_RESET - return the peak memory and reset the counter to start gathering the peak memory data from scratch

Description

The `mkl_peak_mem_usage` function reports the peak memory allocated by the Intel® oneAPI Math Kernel Library (oneMKL) Memory Allocator.

Gathering the peak memory data is turned off by default. If you need to know the peak memory, explicitly turn the data gathering mode on by calling the function with the `MKL_PEAK_MEM_ENABLE` value of the parameter. Use the `MKL_PEAK_MEM` and `MKL_PEAK_MEM_RESET` values only when the data gathering mode is turned on. Otherwise the function returns -1. The data gathering mode leads to performance degradation, so when the mode is turned on, you can turn it off by calling the function with the `MKL_PEAK_MEM_DISABLE` value of the parameter.

NOTE

- If Intel® oneAPI Math Kernel Library (oneMKL) is running in a threaded mode, the `mkl_peak_mem_usage` function may return different amounts of memory from run to run.
- The function reports the peak memory for the entire application, not just for the calling thread.

Return Values

Name	Type	Description
<i>AllocatedBytes</i>	MKL_INT64	The peak memory allocated by the Memory Allocator (in bytes) or -1 in case of errors.

See Also

[Usage Example for the Memory Functions](#)

mkl_malloc

Allocates an aligned memory buffer.

Syntax

```
void* mkl_malloc (size_t alloc_size, int alignment);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>alloc_size</i>	<code>size_t</code>	Size of the buffer to be allocated.
<i>alignment</i>	<code>int</code>	Alignment of the buffer.

Description

The function allocates an *alloc_size*-byte buffer aligned on the *alignment*-byte boundary.

If *alignment* is not a power of 2, the 64-byte alignment is used.

Return Values

Name	Type	Description
<i>a_ptr</i>	void*	Pointer to the allocated buffer if <i>alloc_size</i> ≥ 1, NULL if <i>alloc_size</i> < 1.

See Also

[mkl_free](#)

[Usage Example for the Memory Functions](#)

mkl_calloc

Allocates and initializes an aligned memory buffer.

Syntax

```
void* mkl_calloc (size_t num, size_t size, int alignment);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>num</i>	size_t	The number of elements in the buffer to be allocated.
<i>size</i>	size_t	The size of the element.
<i>alignment</i>	int	Alignment of the buffer.

Description

The function allocates a *num***size*-byte buffer, aligned on the *alignment*-byte boundary, and initializes the buffer with zeros.

If *alignment* is not a power of 2, the 64-byte alignment is used.

Return Values

Name	Type	Description
<i>a_ptr</i>	void*	Pointer to the allocated buffer if <i>size</i> ≥ 1, NULL if <i>size</i> < 1.

See Also

[mkl_malloc](#)

[mkl_realloc](#)

[mkl_free](#)

[Usage Example for the Memory Functions](#)

mkl_realloc

Changes the size of memory buffer allocated by mkl_malloc/mkl_calloc.

Syntax

```
void* mkl_realloc (void *ptr, size_t size);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>ptr</i>	void*	Pointer to the memory buffer allocated by the mkl_malloc or mkl_calloc function or a NULL pointer.
<i>size</i>	size_t	New size of the buffer.

Description

The function changes the size of the memory buffer allocated by the mkl_malloc or mkl_calloc function to *size* bytes. The first bytes of the returned buffer up to the minimum of the old and new sizes keep the content of the input buffer. The returned memory buffer can have a different location than the input one. If *ptr* is NULL, the function works as mkl_malloc.

Return Values

Name	Type	Description
<i>a_ptr</i>	void*	<ul style="list-style-type: none"> • Pointer to the re-allocated buffer if re-allocation is successful. • NULL if re-allocation is unsuccessful.

See Also

mkl_malloc

mkl_calloc

mkl_free

Usage Example for the Memory Functions

mkl_free

Frees the aligned memory buffer allocated by mkl_malloc/mkl_calloc.

Syntax

```
void mkl_free (void *a_ptr);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<code>a_ptr</code>	<code>void*</code>	Pointer to the buffer to be freed.

Description

The function frees the buffer pointed by `a_ptr` and allocated by the `mkl_malloc()` or `mkl_calloc()` function and does nothing if `a_ptr` is `NULL`.

See Also

[mkl_malloc](#)

[mkl_calloc](#)

[Usage Example for the Memory Functions](#)

mkl_set_memory_limit

On Linux, sets the limit of memory that Intel® oneAPI Math Kernel Library (oneMKL) can allocate for a specified type of memory.

Syntax

```
int mkl_set_memory_limit (int mem_type, size_t limit);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>mem_type</code>	<code>int</code>	Type of memory to limit. Possible values: <code>MKL_MEM_MCDRAM</code> - Multi-Channel Dynamic Random Access Memory (MCDRAM).
<code>limit</code>	<code>size_t</code>	Memory limit in megabytes.

Description

This function sets the limit for the amount of memory that Intel® oneAPI Math Kernel Library (oneMKL) can allocate for the specified memory type. The limit bounds both internal allocations (inside Intel® oneAPI Math Kernel Library (oneMKL) computation routines) and external allocations (in a call to `mkl_malloc`, `mkl_calloc`, or `mkl_realloc`). By default no limit is set for memory allocation.

Call `mkl_set_memory_limit` at most once, prior to calling any other Intel® oneAPI Math Kernel Library (oneMKL) function in your application except [mkl_set_interface_layer](#) and [mkl_set_threading_layer](#).

NOTE

- Allocation in MCDRAM requires `libmemkind` and `libjemalloc` dynamic libraries which are a part of Intel® Manycore Platform Software Package (Intel® MPSP) for Linux*.
 - The `mkl_set_memory_limit` function takes precedence over the `MKL_FAST_MEMORY_LIMIT` environment variable.
-

Return Values

Type	Description
int	Status of the function completion: <ul style="list-style-type: none"> • 1 - the limit is set • 0 - the limit is not set

See Also

[mkl_malloc](#)
[mkl_calloc](#)
[mkl_realloc](#)

Usage Example for the Memory Functions

Usage Example for the Memory Functions

```
#include <stdio.h>
#include <mkl.h>

int main(void) {
    double *a, *b, *c;
    int n, i;
    double alpha, beta;
    MKL_INT64 AllocatedBytes;
    int N_AllocatedBuffers;

    alpha = 1.1; beta = -1.2;
    n = 1000;
    mkl_peak_mem_usage(MKL_PEAK_MEM_ENABLE);
    a = (double*)mkl_malloc(n*n*sizeof(double), 64);
    b = (double*)mkl_malloc(n*n*sizeof(double), 64);
    c = (double*)mkl_calloc(n*n, sizeof(double), 64);
    for (i=0; i<(n*n); i++) {
        a[i] = (double)(i+1);
        b[i] = (double)(-i-1);
    }

    dgemm("N", "N", &n, &n, &n, &alpha, a, &n, b, &n, &beta, c, &n);
    AllocatedBytes = mkl_mem_stat(&N_AllocatedBuffers);
    printf("\nDGEMM uses %d bytes in %d buffers", AllocatedBytes, N_AllocatedBuffers);

    mkl_free_buffers();
    mkl_free(a);
    mkl_free(b);
    mkl_free(c);

    AllocatedBytes = mkl_mem_stat(&N_AllocatedBuffers);
    if (AllocatedBytes > 0) {
        printf("\nMKL memory leak!");
        printf("\nAfter mkl_free_buffers there are %d bytes in %d buffers",
            AllocatedBytes, N_AllocatedBuffers);
    }
    printf("\nPeak memory allocated by Intel MKL memory allocator %d bytes. Start to count new memory peak",
        mkl_peak_mem_usage(MKL_PEAK_MEM_RESET));
    a = (double*)mkl_malloc(n*n*sizeof(double), 64);
    a = (double*)mkl_realloc(a, 2*n*n*sizeof(double));
```

```

mkl_free(a);
printf("\nPeak memory allocated by Intel MKL memory allocator after reset of peak memory
counter %d bytes\n",
       mkl_peak_mem_usage(MKL_PEAK_MEM));

return 0;
}

```

Single Dynamic Library Control

Intel® oneAPI Math Kernel Library (oneMKL) provides the Single Dynamic Library (SDL), which enables setting the interface and threading layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time. See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details of SDL and layered model concept. This section describes the functions supporting SDL.

mkl_set_interface_layer

Sets the interface layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time. Use with the Single Dynamic Library.

Syntax

```
int mkl_set_interface_layer (int required_interface);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>required_interface</i>	int	<p>Determines the interface layer. Possible values depend on the system architecture. Some of the values are only available on Linux* OS:</p> <ul style="list-style-type: none"> Intel® 64 architecture: <ul style="list-style-type: none"> MKL_INTERFACE_LP64 for the Intel LP64 interface. MKL_INTERFACE_ILP64 for the Intel ILP64 interface. MKL_INTERFACE_LP64+MKL_INTERFACE_GNU for the GNU* LP64 interface on Linux OS. MKL_INTERFACE_ILP64+MKL_INTERFACE_GNU for the GNU ILP64 interface on Linux OS. IA-32 architecture: <ul style="list-style-type: none"> MKL_INTERFACE_LP64 for the Intel interface on Linux OS. MKL_INTERFACE_LP64+MKL_INTERFACE_GNU or MKL_INTERFACE_GNU for the GNU interface on Linux OS.

Description

If you are using the Single Dynamic Library (SDL), the `mkl_set_interface_layer` function sets the specified interface layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time.

Call this function prior to calling any other Intel® oneAPI Math Kernel Library (oneMKL) function in your application except `mkl_set_threading_layer`. You can call `mkl_set_interface_layer` and `mkl_set_threading_layer` in any order.

The `mkl_set_interface_layer` function takes precedence over the `MKL_INTERFACE_LAYER` environment variable.

See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for the layered model concept and usage details of the SDL.

Return Values

Type	Description
<code>int</code>	<ul style="list-style-type: none"> Current interface layer if it is set in a call to <code>mkl_set_interface_layer</code> or specified by environment variables or defaults. <p>Possible values are specified in Input Parameters.</p> <ul style="list-style-type: none"> -1, if the layer was not specified prior to the call and the input parameter is incorrect.

`mkl_set_threading_layer`

Sets the threading layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time. Use with the Single Dynamic Library (SDL).

Syntax

```
int mkl_set_threading_layer (int required_threading);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>required_threading</code>	<code>int</code>	<p>Determines the threading layer. Possible values:</p> <p><code>MKL_THREADING_INTEL</code> for Intel threading.</p> <p><code>MKL_THREADING_SEQUENTIAL</code> for the sequential mode of Intel® oneAPI Math Kernel Library (oneMKL).</p> <p><code>MKL_THREADING_TBB</code> for threading with the Intel® Threading Building Blocks.</p> <p><code>MKL_THREADING_PGI</code> for PGI threading on Windows* or Linux* operating system only. Do not use this value with the SDL for Intel® Many Integrated Core (Intel® MIC) Architecture.</p>

NOTE PGI* support is deprecated and will be removed in the oneMKL 2025.0 release.

Name	Type	Description
		MKL_THREADING_GNU for GNU threading on Linux* operating system only. Do not use this value with the SDL for Intel MIC Architecture.

Description

If you are using the Single Dynamic Library (SDL), the `mkl_set_threading_layer` function sets the specified threading layer for Intel® oneAPI Math Kernel Library (oneMKL) at run time.

Call this function prior to calling any other Intel® oneAPI Math Kernel Library (oneMKL) function in your application except `mkl_set_interface_layer`.

You can call `mkl_set_threading_layer` and `mkl_set_interface_layer` in any order.

The `mkl_set_threading_layer` function takes precedence over the `MKL_THREADING_LAYER` environment variable.

See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for the layered model concept and usage details of the SDL.

Return Values

Type	Description
<code>int</code>	<ul style="list-style-type: none"> Current threading layer if it is set in a call to <code>mkl_set_threading_layer</code> or specified by environment variables or defaults. Possible values are specified in Input Parameters. -1, if the layer was not specified prior to the call and the input parameter is incorrect.

`mkl_set_xerbla`

Replaces the error handling routine. Use with the Single Dynamic Library .

Syntax

```
XerblaEntry mkl_set_xerbla (XerblaEntry new_xerbla_ptr);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>new_xerbla_ptr</code>	<code>XerblaEntry</code>	Pointer to the error handling routine to be used.

Description

The `mkl_set_xerbla` function replaces the error handling routine that is called by Intel® oneAPI Math Kernel Library (oneMKL) functions with the routine specified by the parameter.

See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details about SDL.

Return Values

The function returns the pointer to the replaced error handling routine.

See Also

[xerbla](#)

mkl_set_progress

Replaces the progress information routine.

Syntax

```
ProgressEntry mkl_set_progress (ProgressEntry new_progress_ptr);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>new_progress_ptr</code>	<code>ProgressEntry</code>	Pointer to the progress information routine to be used.

Description

The `mkl_set_progress` function replaces the currently used progress information routine with the routine specified by the parameter.

Usually a user-supplied `mkl_progress` function redefines the default `mkl_progress` function automatically. However, you must call `mkl_set_progress` to replace the default `mkl_progress` on Windows* in any of the following cases:

- You are using the Single Dynamic Library (SDL) `mkl_rt.lib`.
- You link dynamically with ScaLAPACK.

See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for details of SDL.

Return Values

The function returns the pointer to the replaced progress information routine.

See Also

[mkl_progress](#)

mkl_set_pardiso_pivot

Replaces the routine handling Intel® oneAPI Math Kernel Library (oneMKL) PARDISO pivots with a user-defined routine. Use with the Single Dynamic Library (SDL).

Syntax

```
PardisopivotEntry mkl_set_pardiso_pivot (PardisopivotEntry new_pardiso_pivot_ptr);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>new_pardiso_pivot_ptr</code>	<code>PardisoPivotEntry</code>	Pointer to the pivot setting routine to be used.

Description

If you are using the Single Dynamic Library (SDL), the `mkl_set_pardiso_pivot` function replaces the pivot setting routine that is called by Intel® oneAPI Math Kernel Library (oneMKL) functions with the routine specified by the parameter.

See *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for usage details of the SDL.

Return Values

Type	Description
<code>PardisoPivotEntry</code>	Pointer to the replaced pivot setting routine.

See Also

[mkl_pardiso_pivot](#)

Conditional Numerical Reproducibility Control

The CNR mode of Intel® oneAPI Math Kernel Library (oneMKL) ensures bitwise reproducible results from run to run of Intel® oneAPI Math Kernel Library (oneMKL) functions on a fixed number of threads for a specific Intel instruction set architecture (ISA) under the following conditions:

- Calls to Intel® oneAPI Math Kernel Library (oneMKL) occur in a single executable
- The number of computational threads used by the library does not change in the run

Intel® oneAPI Math Kernel Library (oneMKL) offers both functions and environment variables to support conditional numerical reproducibility. See the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide* for more information on bitwise reproducible results of computations and for details about the environment variables.

The support functions enable you to configure the CNR mode and also provide information on the current and optimal CNR branch on your system. [Usage Examples for CNR Support Functions](#) illustrate usage of these functions.

Important

Call the functions that define the behavior of CNR before any of the math library functions that they control.

Intel® oneAPI Math Kernel Library (oneMKL) provides named constants for use as input and output parameters of the functions instead of integer values. See [Named Constants for CNR Control](#) for a list of the named constants.

Although you can configure the CNR mode using either the support functions or the environment variables, the functions offer more flexible configuration and control than the environment variables. Settings specified by the functions take precedence over the settings specified by the environment variables.

Use Intel® oneAPI Math Kernel Library (oneMKL) in the CNR mode only in case a need for bitwise reproducible results is critical. Otherwise, run Intel® oneAPI Math Kernel Library (oneMKL) as usual to avoid performance degradation.

While you can supply unaligned input and output data to Intel® oneAPI Math Kernel Library (oneMKL) functions running in the CNR mode, use of aligned data is recommended. Refer to [Reproducibility Conditions](#) for more details.

Product and Performance Information

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

Notice revision #20201201

mkl_cbwr_set

Configures the CNR mode of Intel® oneAPI Math Kernel Library (oneMKL).

Syntax

```
int mkl_cbwr_set (int setting);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>setting</i>	int	CNR branch to set. See Named Constants for CNR Control for a list of named constants that specify the settings.

Description

The `mkl_cbwr_set` function configures the CNR mode. In this release, it sets the CNR branch and turns on the CNR mode.

NOTE

Settings specified by the `mkl_cbwr_set` function take precedence over the settings specified by the `MKL_CBWR` environment variable.

Return Values

Name	Type	Description
<i>status</i>	int	The status of the function completion: <ul style="list-style-type: none">• <code>MKL_CBWR_SUCCESS</code> - the function completed successfully.• <code>MKL_CBWR_ERR_INVALID_INPUT</code> - an invalid setting is requested.• <code>MKL_CBWR_ERR_UNSUPPORTED_BRANCH</code> - the input value of the branch does not match the instruction set architecture (ISA) of your system. See Named Constants for CNR Control for more details.

Name	Type	Description
		<ul style="list-style-type: none">• <code>MKL_CBWR_ERR_MODE_CHANGE_FAILURE</code> - the <code>mkl_cbwr_set</code> function requested to change the current CNR branch after a call to some Intel® oneAPI Math Kernel Library (oneMKL) function other than a CNR function.

See Also

[Usage Examples for CNR Support Functions](#)

`mkl_cbwr_get`

Returns the current CNR settings.

Syntax

```
int mkl_cbwr_get (int option);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>option</i>	int	<p>Specifies the CNR settings requested. Named constants define possible values of <i>option</i>:</p> <ul style="list-style-type: none">• <code>MKL_CBWR_BRANCH</code> - returns the current CNR branch only.• <code>MKL_CBWR_ALL</code> - returns all CNR settings including strict CNR setting.

Description

The `mkl_cbwr_get` function returns the requested CNR settings. The function returns `MKL_CBWR_ERR_INVALID_INPUT` if an invalid option is specified.

NOTE

To enable CNR mode, use the `mkl_cbwr_set` function or environment variables. For more details, see the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

Return Values

Name	Type	Description
<i>setting</i>	int	<p>Requested CNR settings. See Named Constants for CNR Control for a list of named constants that specify the settings.</p> <p>If the value of the <i>option</i> parameter is not permitted, contains the <code>MKL_CBWR_ERR_INVALID_INPUT</code> error code.</p>

See Also

Usage Examples for CNR Support Functions

`mkl_cbwr_set`

`mkl_cbwr_get_auto_branch`

Automatically detects the CNR code branch for your platform.

Syntax

```
int mkl_cbwr_get_auto_branch (void);
```

Include Files

- `mkl.h`

Description

The `mkl_cbwr_get_auto_branch` function uses a run-time CPU check to return a CNR branch that is optimized for the processor where the program is currently running.

Product and Performance Information

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

Notice revision #20201201

Return Values

Name	Type	Description
<code>setting</code>	<code>int</code>	Automatically detected CNR branch. May be any <i>specific</i> branch listed in Named Constants for CNR Control .

See Also

Usage Examples for CNR Support Functions

Named Constants for CNR Control

Use the conditional numerical reproducibility (CNR) functionality in Intel® oneAPI Math Kernel Library (oneMKL) to obtain reproducible results from MKL routines. When enabling CNR, you choose a specific code branch of Intel® oneAPI Math Kernel Library (oneMKL) that corresponds to the instruction set architecture (ISA) that you target. Use these named constants to specify the code branch and other CNR options.

Named Constant	Value	Description
CNR Branches		
<code>MKL_CBWR_OFF</code>	0	Disable CNR mode
<code>MKL_CBWR_BRANCH_OFF</code>	1	CNR mode is disabled
<code>MKL_CBWR_AUTO</code>	2	Choose branch automatically. CNR mode uses the standard ISA-based dispatching model while ensuring fixed cache sizes, deterministic reductions, and static scheduling
<code>MKL_CBWR_COMPATIBLE</code>	3	Intel® Streaming SIMD Extensions 2 (Intel® SSE2) without <code>rcpps/rsqrtps</code> instructions

Named Constant	Value	Description
MKL_CBWR_SSE2	4	Intel SSE2
MKL_CBWR_SSE3	5	DEPRECATED. Intel® Streaming SIMD Extensions 3 (Intel® SSE3). This setting is kept for backward compatibility and is equivalent to MKL_CBWR_SSE2.
MKL_CBWR_SSSE3	6	Supplemental Streaming SIMD Extensions 3 (SSSE3)
MKL_CBWR_SSE4_1	7	Intel® Streaming SIMD Extensions 4-1 (SSE4-1)
MKL_CBWR_SSE4_2	8	Intel® Streaming SIMD Extensions 4-2 (SSE4-2)
MKL_CBWR_AVX	9	Intel® Advanced Vector Extensions (Intel® AVX)
MKL_CBWR_AVX2	10	Intel® Advanced Vector Extensions 2 (Intel® AVX2)
MKL_CBWR_AVX512_MIC	11	DEPRECATED. Intel® Advanced Vector Extensions 512 (Intel® AVX-512) on Intel® Xeon Phi™ processors. This setting is kept for backward compatibility and is equivalent to MKL_CBWR_AVX2.
MKL_CBWR_AVX512	12	Intel AVX-512 on Intel® Xeon® processors
MKL_CBWR_AVX512_MIC_E1	13	DEPRECATED. Intel® Advanced Vector Extensions 512 (Intel® AVX-512) for Intel® Many Integrated Core Architecture (Intel® MIC Architecture) with support of AVX512_4FMAPS and AVX512_4VNNIW instruction groups enabled processors. This setting is kept for backward compatibility and is equivalent to MKL_CBWR_AVX2.
MKL_CBWR_AVX512_E1	14	Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support of Vector Neural Network Instructions enabled processors
CNR Flags		
MKL_CBWR_STRICT	65536 or 0x10000	Strict CNR mode enabled. See Reproducibility Conditions for more information.

When specifying the CNR branch with the named constants, be aware of the following:

- Reproducible results are provided under [Reproducibility Conditions](#).
- Settings other than MKL_CBWR_AUTO or MKL_CBWR_COMPATIBLE are available only for Intel processors.
- Intel and Intel compatible CPUs have a few instructions, such as approximation instructions rcpps/rsqrtps, that may return different results. Setting the branch to MKL_CBWR_COMPATIBLE ensures that Intel® oneAPI Math Kernel Library (oneMKL) does not use these instructions and forces a single Intel SSE2-only code path to be executed.

Product and Performance Information

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

Notice revision #20201201

See Also

[Usage Examples for CNR Support Functions](#)

Reproducibility Conditions

To get reproducible results from run to run, ensure that the number of threads is fixed and constant. Specifically:

- If you are running your program with OpenMP* parallelization on different processors, explicitly specify the number of threads.
- To ensure that your application has deterministic behavior with OpenMP* parallelization and does not adjust the number of threads dynamically at run time, set `MKL_DYNAMIC` and `OMP_DYNAMIC` to `FALSE`. This is especially needed if you are running your program on different systems.
- If you are running your program with the Intel® Threading Building Blocks parallelization, numerical reproducibility is not guaranteed.

Strict CNR Mode

In strict CNR mode, oneAPI Math Kernel Library provides bitwise reproducible results for a limited set of functions and code branches even when the number of threads changes. These routines and branches support strict CNR mode (64-bit libraries only):

- `?gemm`, `?symm`, `?hemm`, `?trsm`, and their CBLAS equivalents (`cblas_?gemm`, `cblas_?symm`, `cblas_?hemm`, and `cblas_?trsm`).
- Intel® Advanced Vector Extensions 2 (Intel® AVX2) or Intel® Advanced Vector Extensions 512 (Intel® AVX-512).

When using other routines or CNR branches, oneAPI Math Kernel Library operates in standard (non-strict) CNR mode, subject to the restrictions described above. Enabling strict CNR mode can reduce performance.

NOTE

- As usual, you should align your data, even in CNR mode, to obtain the best possible performance. While CNR mode also fully supports unaligned input and output data, the use of it might reduce the performance of some oneAPI Math Kernel Library functions on earlier Intel processors. To ensure proper alignment of arrays, allocate memory for them using `mkl_malloc/mkl_calloc`.
- Conditional Numerical Reproducibility does not ensure that bitwise-identical NaN values are generated when the input data contains NaN values.
- If dynamic memory allocation fails on one run but succeeds on another run, you may fail to get reproducible results between these two runs.

Product and Performance Information

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

Notice revision #20201201

See Also

`mkl_malloc`

`mkl_calloc`

Usage Examples for CNR Support Functions

The following examples illustrate usage of support functions for conditional numerical reproducibility.

Setting Automatically Detected CNR Branch

```
#include <mkl.h>
int main(void) {
    int my_cbwr_branch;
```

```

/* Find the available MKL_CBWR_BRANCH automatically */
my_cbwr_branch = mkl_cbwr_get_auto_branch();
/* User code without Intel MKL calls */
/* Piece of the code where CNR of Intel MKL is needed */
/* The performance of Intel MKL functions might be reduced for CNR mode */
if (mkl_cbwr_set(my_cbwr_branch)!=MKL_CBWR_SUCCESS) {
    printf("Error in setting MKL_CBWR_BRANCH! Aborting...\n");
    return;
}
/* CNR calls to Intel MKL + any other code */
}

```

Use of the mkl_cbwr_get Function

```

#include <mkl.h>
int main(void) {
    int my_cbwr_branch;
    /* Piece of the code where CNR of Intel MKL is analyzed */
    my_cbwr_branch = mkl_cbwr_get(MKL_CBWR_BRANCH);
    switch (my_cbwr_branch) {
        case MKL_CBWR_AUTO:
            /* actions in case of automatic mode */
            break;
        case MKL_CBWR_SSSE3:
            /* actions for SSSE3 code */
            break;
        default:
            /* all other cases */
    }
    /* User code */
}

```

Product and Performance Information

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

Notice revision #20201201

Miscellaneous

mkl_progress

Provides progress information.

Syntax

```
int mkl_progress (int* thread_process, int* step, char* stage, int lstage);
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<code>thread_process</code>	<code>const int*</code>	Indicates the number of thread or process the progress routine is called from: <ul style="list-style-type: none"> The thread number for non-cluster components linked with OpenMP threading layer Zero for non-cluster components linked with sequential threading layer The process number (MPI rank) for cluster components
<code>step</code>	<code>const int*</code>	Pointer to the linear progress indicator that shows the amount of work done. Increases from 0 to the linear size of the problem during the computation.
<code>stage</code>	<code>const char*</code>	Message indicating the name of the routine or the name of the computation stage the progress routine is called from.
<code>lstage</code>	<code>int</code>	The length of a stage string excluding the trailing NULL character.

Description

The `mkl_progress` function is intended to track progress of a lengthy computation and/or interrupt the computation. By default this routine does nothing but the user application can redefine it to obtain the computation progress information. You can set it to perform certain operations during the routine computation, for instance, to print a progress indicator. A non-zero return value may be supplied by the redefined function to break the computation.

NOTE

The user-defined `mkl_progress` function must be thread-safe.

Some Intel® oneAPI Math Kernel Library (oneMKL) functions from LAPACK, ScaLAPACK, DSS/PARDISO, and Parallel Direct Sparse Solver for Clusters regularly call the `mkl_progress` function during the computation. Refer to the description of a specific function from those domains to see whether the function supports this feature or not.

If a LAPACK function returns `info=-1002`, the function was interrupted by `mkl_progress`. Because ScaLAPACK does not support interruption of the computation, Intel® oneAPI Math Kernel Library (oneMKL) ignores any value returned by `mkl_progress`.

While a user-supplied `mkl_progress` function usually redefines the default `mkl_progress` function automatically, some configurations require calling the `mkl_set_progress` function to replace the default `mkl_progress` function. Call `mkl_set_progress` to replace the default `mkl_progress` on Windows* in any of the following cases:

- You are using the Single Dynamic Library (SDL) `mkl_rt.lib`.
- You link dynamically with ScaLAPACK.

Warning

The `mkl_progress` function supports OpenMP*/TBB threading and sequential execution for specific routines.

Return Values

Name	Type	Description
<code>stopflag</code>	<code>int</code>	The stopping flag. A non-zero flag forces the routine to be interrupted. The zero flag is the default return value.

Example

The following example prints the progress information to the standard output device:

```
#include <stdio.h>
#include <string.h>
#define BUFLLEN 16
int mkl_progress( int* thread_process, int* step, char* stage, int lstage )
{
    char buf[BUFLLEN];
    if( lstage >= BUFLLEN ) lstage = BUFLLEN-1;
    strncpy( buf, stage, lstage );
    buf[lstage] = '\0';
    printf( "In thread %i, at stage %s, steps passed %i\n", *thread_process, buf, *step );
    return 0;
}
```

mkl_enable_instructions

Enables dispatching for new Intel® architectures or restricts the set of Intel® instruction sets available for dispatching. The `mkl_enable_instructions` function must be called only once, before any other Intel® oneAPI Math Kernel Library (oneMKL) functions.

Syntax

```
int mkl_enable_instructions (int isa);
```

Input Parameters

Name	Type	Description
<code>isa</code>	<code>int</code>	The latest Intel® instruction-set architecture (ISA) for Intel® oneAPI Math Kernel Library (oneMKL) to dispatch.
<code>MKL_ENABLE_AVX512</code>		Intel® Advanced Vector Extensions 512 (Intel® AVX-512)
<code>MKL_ENABLE_AVX512_E1</code>		Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for Intel® Deep Learning Boost (Intel® DL Boost).
<code>MKL_ENABLE_AVX512_E2</code>		Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for Intel® Deep Learning Boost (Intel® DL

Name	Type	Description
		Boost), EVEX-encoded AES, and Carry-Less Multiplication Quadword instructions
	MKL_ENABLE_AVX512_E3	Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for Intel® Deep Learning Boost (Intel® DL Boost) and bfloat16
	MKL_ENABLE_AVX512_E4	Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for INT8, BF16, FP16 (limited) instructions, and Intel® Advanced Matrix Extensions (Intel® AMX) with INT8 and BF16
	MKL_ENABLE_AVX512_E5	Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for INT8, BF16, FP16 (limited) instructions, and Intel® Advanced Matrix Extensions (Intel® AMX) with INT8, BF16, and FP16
<hr/> NOTE Not dispatched by default. <hr/>		
	MKL_ENABLE_AVX2	Intel® Advanced Vector Extensions 2 (Intel® AVX2)
	MKL_ENABLE_AVX2_E1	Intel® Advanced Vector Extensions 2 (Intel® AVX2) with support for Intel® Deep Learning Boost (Intel® DL Boost)
	MKL_ENABLE_SSE4_2	Intel® Streaming SIMD Extensions 4.2 (Intel® SSE4.2)

Description

Intel® oneAPI Math Kernel Library (oneMKL) does run-time processor dispatching to identify appropriate internal code paths to traverse for Intel® oneAPI Math Kernel Library (oneMKL) functions called by the application. The `mkl_enable_instructions` function controls the behavior of the dispatcher to do either of the following:

- Enable dispatching for new Intel architectures.

Intel® oneAPI Math Kernel Library (oneMKL) does not dispatch instruction sets that do not have silicon available at time of the product launch. Call `mkl_enable_instructions` to enable dispatching the code path for such an ISA in a simulator environment or on hardware that supports this ISA.

- Restrict the set of Intel instruction sets available for dispatching.

Call `mkl_enable_instructions` to restrict dispatching to code paths for earlier ISA. For example, if the hardware supports Intel AVX, a call to `mkl_enable_instructions` with the `MKL_ENABLE_SSE4_2` parameter forces the dispatcher to use the Intel SSE4-2 code path.

If the system does not support the instruction set specified by the `isa` parameter or if the system is based on a non-Intel architecture, `mkl_enable_instructions` does nothing and returns zero.

Settings specified by the `mkl_enable_instructions` function set an upper limit to settings specified by the `mkl_cbwr_set` function.

You can use the `MKL_ENABLE_INSTRUCTIONS` environment variable instead of calling `mkl_enable_instructions` (for more details, see the [Intel® oneAPI Math Kernel Library \(oneMKL\) Developer Guide](#)); however, the settings specified by the function take precedence over the settings specified by the environment variable.

Return Values

Name	Type	Description
<code>irc</code>	<code>int</code>	<p>Function completion status:</p> <p>1 - Intel® oneAPI Math Kernel Library (oneMKL) dispatches the code path for the specified ISA by default.</p> <p>0 - The request is rejected. Usually this occurs if <code>mkl_enable_instructions</code> was called:</p> <ul style="list-style-type: none"> • After another Intel® oneAPI Math Kernel Library (oneMKL) function • On a non-Intel architecture • With an incompatible ISA specified

Product and Performance Information

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

Notice revision #20201201

`mkl_set_env_mode`

Sets up the mode that ignores environment settings specific to Intel® oneAPI Math Kernel Library (oneMKL).

Syntax

```
int mkl_set_env_mode(int mode);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>mode</code>	<code>int</code>	Specifies what mode to set. For details, see Description . Possible values: <ul style="list-style-type: none"> 0 - Do nothing. Use this value to query the current environment mode. 1 - Make Intel® oneAPI Math Kernel Library (oneMKL) ignore environment settings specific to the library.

Description

In the default *environment mode*, Intel® oneAPI Math Kernel Library (oneMKL) can control its behavior using environment variables for threading, memory management, Conditional Numerical Reproducibility, automatic offload, and so on. The `mkl_set_env_mode` function sets up the environment mode that ignores all settings specified by Intel® oneAPI Math Kernel Library (oneMKL) environment variables except `MIC_LD_LIBRARY_PATH` and `MKLROOT`.

Return Values

Name	Type	Description
<code>current_mode</code>	<code>int</code>	Environment mode that was used before the function call: <ul style="list-style-type: none"> 0 - Default 1 - Ignore environment settings specific to Intel® oneAPI Math Kernel Library (oneMKL).

mkl_verbose

Enables or disables Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode.

Syntax

```
int mkl_verbose (int enable);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>enable</code>	<code>int</code>	Desired state of the Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode. Indicates whether printing Intel® oneAPI Math Kernel Library (oneMKL) function call information should be turned on or off. Possible values: <ul style="list-style-type: none"> 0 – disable the Verbose mode 1 – enable the Verbose mode (GPU application: enable the Verbose mode without timing) 2 – enable the Verbose mode (GPU application: enable the Verbose mode with synchronous timing)

Description

This function enables or disables the Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode, in which computational functions print call description information. For details of the Verbose mode, see the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*, available in the Intel® Software Documentation Library.

NOTE

The setting for the Verbose mode specified by the `mkl_verbose` function takes precedence over the setting specified by the `MKL_VERBOSE` environment variable.

Return Values

Name	Type	Description
<code>status</code>	<code>int</code>	<ul style="list-style-type: none"> If the requested operation completed successfully, contains previous state of the verbose mode: <ul style="list-style-type: none"> 0 – Verbose mode was disabled 1 – Verbose mode was enabled (GPU application: Verbose mode was enabled without timing) 2 – Verbose mode was enabled (GPU application: Verbose mode was enabled with synchronous timing) If the function failed to complete the operation because of an incorrect input parameter, equals -1.

See Also

[Intel Software Documentation Library](#)

`mkl_verbose_output_file`

Write output in Intel® oneAPI Math Kernel Library (oneMKL) Verbose mode to a file.

Syntax

```
int mkl_verbose_output_file (const char*filename);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>filename</code>	<code>char</code>	Name of file. Specify the complete path of the output file.

Description

This function writes the output in Verbose mode to the file specified in the path.

If the write operation is successful, the function returns 0.

If the file does not exist or cannot be opened, the write operation is unsuccessful. The function returns 1 and defaults to `mkl_verbose` behavior by printing to `stdout`.

NOTE

You can alternatively use MKL_VERBOSE_OUTPUT_FILE environment variable instead of calling the `mkl_verbose_output_file` function. If you want to use the environment variable option, you must set it to the complete path of the output file.

Important The setting for the verbose output file specified by the `mkl_verbose_output_file` function takes precedence over the setting specified by the MKL_VERBOSE_OUTPUT_FILE environment variable.

For more information on the Verbose mode, see the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*, available in the Intel® Software Documentation Library.

Return Values

Name	Type	Description
<code>status</code>	<code>int</code>	<ul style="list-style-type: none"> 0 indicates that the write operation was successful. 1 indicates that the write operation was unsuccessful.

See Also

[Intel Software Documentation Library](#)

mkl_set_mpi

Sets the implementation of the message-passing interface to be used by Intel® oneAPI Math Kernel Library (oneMKL).

Syntax

```
int mkl_set_mpi (int vendor, const char *custom_library_name);
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>vendor</code>	<code>int</code>	<p>Specifies the implementation of the message-passing interface (MPI) to use:</p> <p>Possible values:</p> <ul style="list-style-type: none"> MKL_BLACS_CUSTOM - a custom MPI library. Requires a prebuilt custom MPI BLACS library. MKL_BLACS_MSMPPI - Microsoft MPI library. MKL_BLACS_INTELMPI - Intel® MPI library. MKL_BLACS_MPICH - MPICH MPI library.

Name	Type	Description
<code>custom_library_name</code>	<code>const char *</code>	The filename (without a directory name) of the custom BLACS dynamic library to use. This library must be located in the directory with your application executable or with Intel® oneAPI Math Kernel Library (oneMKL) dynamic libraries. Can be <code>NULL</code> or an empty string.

Description

Call this function to set the MPI implementation to be used by Intel® oneAPI Math Kernel Library (oneMKL) on Windows* OS when dynamic Intel® oneAPI Math Kernel Library (oneMKL) libraries are used. For all other configurations, the function returns an error indicating that you cannot set the MPI implementation. You can specify your own prebuilt dynamic BLACS library for a custom MPI by setting `vendor` to `MKL_BLACS_CUSTOM` and optionally passing the name of the custom BLACS dynamic library. If the `custom_library_name` parameter is `NULL` or an empty string, Intel® oneAPI Math Kernel Library (oneMKL) uses the default platform-specific library name: `mkl_blacs_custom_lp64.dll` or `mkl_blacs_custom_ilp64.dll`, depending on whether the BLACS interface linked against your application is LP64 or ILP64.

Return Values

Name	Type	Description
<code>status</code>	<code>int</code>	<p>The return status:</p> <ul style="list-style-type: none"> • 0 - The function completed successfully. • -1 - The <code>vendor</code> parameter is invalid. • -2 - The <code>custom_library_name</code> parameter is invalid. • -3 - The MPI library cannot be set at this point.

Product and Performance Information

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

Notice revision #20201201

mkl_finalize

Terminates Intel® oneAPI Math Kernel Library (oneMKL) execution environment and frees resources allocated by the library.

Syntax

```
void mkl_finalize(void);
```

Include Files

- `mkl.h`

Description

This function frees resources allocated by Intel® oneAPI Math Kernel Library (oneMKL). Once this function is called, the application can no longer call Intel® oneAPI Math Kernel Library (oneMKL) functions other than `mkl_finalize`.

In particular, the `mkl_finalize` function enables you to free resources when a third-party shared library is statically linked to Intel® oneAPI Math Kernel Library (oneMKL). To avoid resource leaks that may happen when a shared library is loaded and unloaded multiple times, call `mkl_finalize` each time the library is unloaded. The recommended method to do this depends on the operating system:

- On Linux* or macOS*, place the call into a shared library destructor.
- On Windows*, call `mkl_finalize` from the `DLL_PROCESS_DETACH` handler of `DllMain`.

NOTE

Intel® oneAPI Math Kernel Library (oneMKL) shared libraries automatically perform finalization when they are unloaded. If an application is statically linked to Intel® oneAPI Math Kernel Library (oneMKL), the operating system frees all resources allocated by Intel® oneAPI Math Kernel Library (oneMKL) during termination of the process associated with the application.

BLACS Routines

Intel® oneAPI Math Kernel Library implements FORTRAN 77 routines from the BLACS (Basic Linear Algebra Communication Subprograms) package. These routines are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The BLACS routines make linear algebra applications both easier to program and more portable. For this purpose, they are used in Intel® oneAPI Math Kernel Library (oneMKL) intended for the Linux* and Windows* OSs as the communication layer of ScaLAPACK and Cluster FFT.

On computers, a linear algebra matrix is represented by a two dimensional array (2D array), and therefore the BLACS operate on 2D arrays. See description of the basic [matrix shapes](#) in a special topic.

The BLACS routines implemented in Intel® oneAPI Math Kernel Library (oneMKL) are of four categories:

- Combines
- Point to Point Communication
- Broadcast
- Support.

The [Combines](#) take data distributed over processes and combine the data to produce a result. The [Point to Point](#) routines are intended for point-to-point communication and [Broadcast](#) routines send data possessed by one process to all processes within a scope.

The [Support routines](#) perform distinct tasks that can be used for initialization, destruction, information, and miscellaneous tasks.

Product and Performance Information

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

Notice revision #20201201

Matrix Shapes

The BLACS routines recognize the two most common classes of matrices for dense linear algebra. The first of these classes consists of general rectangular matrices, which in machine storage are 2D arrays consisting of m rows and n columns, with a leading dimension, lda , that determines the distance between successive columns in memory.

The *general rectangular* matrices take the following parameters as input when determining what array to operate on:

m	(input) INTEGER. The number of matrix rows to be operated on.
n	(input) INTEGER. The number of matrix columns to be operated on.
a	(input/output) TYPE (depends on routine), array of dimension (lda, n) . A pointer to the beginning of the (sub)array to be sent.
lda	(input) INTEGER. The distance between two elements in matrix row.

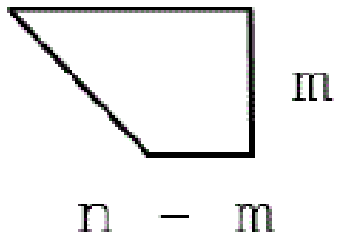
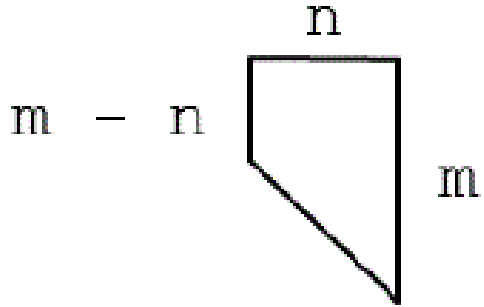
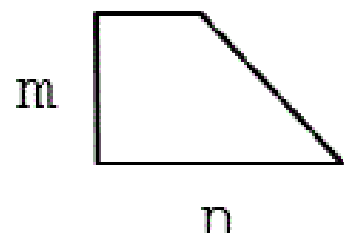
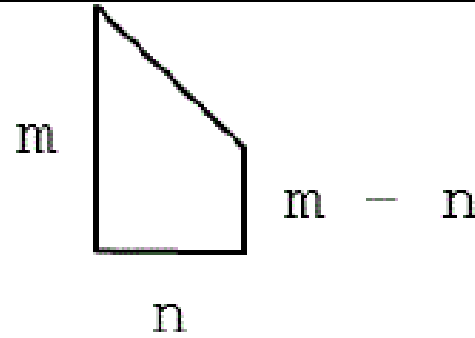
The second class of matrices recognized by the BLACS are *trapezoidal* matrices (triangular matrices are a sub-class of trapezoidal). Trapezoidal arrays are defined by m , n , and lda , as above, but they have two additional parameters as well. These parameters are:

$uplo$	(input) CHARACTER*1 . Indicates whether the matrix is upper or lower trapezoidal, as discussed below.
$diag$	(input) CHARACTER*1 . Indicates whether the diagonal of the matrix is unit diagonal (will not be operated on) or otherwise (will be operated on).

The shape of the trapezoidal arrays is determined by these parameters as follows:

__border__top

Trapezoidal Arrays Shapes

$uplo$	$m \leq n$	$m > n$
"U"		
"L"		

The packing of arrays, if required, so that they may be sent efficiently is hidden, allowing the user to concentrate on the logical matrix, rather than on how the data is organized in the system memory.

Repeatability and Coherence

Floating point computations are not exact on almost all modern architectures. This lack of precision is particularly problematic in parallel operations. Since floating point computations are inexact, algorithms are classified according to whether they are *repeatable* and to what degree they guarantee *coherence*.

- Repeatable: a routine is repeatable if it is guaranteed to give the same answer if called multiple times with the same parallel configuration and input.
- Coherent: a routine is coherent if all processes selected to receive the answer get identical results.

NOTE

Repeatability and coherence do not effect correctness. A routine may be both incoherent and non-repeatable, and still give correct output. But inaccuracies in floating point calculations may cause the routine to return differing values, all of which are equally valid.

Repeatability

Because the precision of floating point arithmetic is limited, it is not truly associative: $(a + b) + c$ might not be the same as $a + (b + c)$. The lack of exact arithmetic can cause problems whenever the possibility for reordering of floating point calculations exists. This problem becomes prevalent in parallel computing due to race conditions in message passing. For example, consider a routine which sums numbers stored on different processes. Assume this routine runs on four processes, with the numbers to be added being the process numbers themselves. Therefore, process 0 has the value 0:0, process 1 has the value 1:0, and so on.

One algorithm for the computation of this result is to have all processes send their process numbers to process 0; process 0 adds them up, and sends the result back to all processes. So, process 0 would add a number to 0:0 in the first step. If receiving the process numbers is ordered so that process 0 always receives the message from process 1 first, then 2, and finally 3, this results in a repeatable algorithm, which evaluates the expression $((0:0 + 1:0) + 2:0) + 3:0$.

However, to get the best parallel performance, it is better not to require a particular ordering, and just have process 0 add the first available number to its value and continue to do so until all numbers have been added in. Using this method, a race condition occurs, because the order of the operation is determined by the order in which process 0 receives the messages, which can be effected by any number of things. This implementation is not repeatable, because the answer can vary between invocations, even if the input is the same. For instance, one run might produce the sequence $((0:0 + 1:0) + 2:0) + 3:0$, while a subsequent run could produce $((0:0 + 2:0) + 1:0) + 3:0$. Both of these results are correct summations of the given numbers, but because of floating point roundoff, they might be different.

Coherence

A routine produces coherent output if all processes are guaranteed to produce the exact same results. Obviously, almost no algorithm involving communication is coherent if communication can change the values being communicated. Therefore, if the parallel system being studied cannot guarantee that communication between processes preserves values, no routine is guaranteed to produce coherent results.

If communication is assumed to be coherent, there are still various levels of coherent algorithms. Some algorithms guarantee coherence only if floating point operations are done in the exact same order on every node. This is *homogeneous coherence*: the result will be coherent if the parallel machine is homogeneous in its handling of floating point operations.

A stronger assertion of coherence is *heterogeneous coherence*, which does not require all processes to have the same handling of floating point operations.

In general, a routine that is homogeneous coherent performs computations redundantly on all nodes, so that all processes get the same answer only if all processes perform arithmetic in the exact same way, whereas a routine which is heterogeneous coherent is usually constrained to having one process calculate the final result, and broadcast it to all other processes.

Example of Incoherence

An incoherent algorithm is one which does not guarantee that all processes get the same result even on a homogeneous system with coherent communication. The previous example of summing the process numbers demonstrates this kind of behavior. One way to perform such a sum is to have every process broadcast its number to all other processes. Each process then adds these numbers, starting with its own. The calculations performed by each process receives would then be:

- Process 0 : $((0:0 + 1:0) + 2:0) + 3:0$
- Process 1 : $((1:0 + 2:0) + 3:0) + 0:0$
- Process 2 : $((2:0 + 3:0) + 0:0) + 1:0$
- Process 3 : $((3:0 + 0:0) + 1:0) + 0:0$

All of these results are equally valid, and since all the results might be different from each other, this algorithm is incoherent. Notice, however, that this algorithm is repeatable: each process will get the same result if the algorithm is called again on the same data.

Example of Homogeneous Coherence

Another way to perform this summation is for all processes to send their data to all other processes, and to ensure the result is not incoherent, enforce the ordering so that the calculation each node performs is $((0:0 + 1:0) + 2:0) + 3:0$. This answer is the same for all processes only if all processes do the floating point arithmetic in the same way. Otherwise, each process may make different floating point errors during the addition, leading to incoherence of the output. Notice that since there is a specific ordering to the addition, this algorithm is repeatable.

Example of Heterogeneous Coherence

In the final example, all processes send the result to process 0, which adds the numbers and broadcasts the result to the rest of the processes. Since one process does all the computation, it can perform the operations in any order and it will give coherent results as long as communication is itself coherent. If a particular order is not forced on the the addition, the algorithm will not be repeatable. If a particular order is forced, it will be repeatable.

Summary

Repeatability and coherence are separate issues which may occur in parallel computations. These concepts may be summarized as:

- Repeatability: The routine will yield the exact same result if it run multiple times on an identical problem. Each process may get a different result than the others (i.e., repeatability does not imply coherence), but that value will not change if the routine is invoked multiple times.
- Homogeneous coherence: All processes selected to possess the result will receive the exact same answer if:
 - Communication does not change the value of the communicated data.
 - All processes perform floating point arithmetic exactly the same.
- Heterogeneous coherence: All processes will receive the exact same answer if communication does not change the value of the communicated data.

In general, lack of the associative property for floating point calculations may cause both incoherence and non-repeatability. Algorithms that rely on redundant computations are at best homogeneous coherent, and algorithms in which one process broadcasts the result are heterogeneous coherent. Repeatability does not imply coherence, nor does coherence imply repeatability.

Since these issues do not effect the correctness of the answer, they can usually be ignored. However, in very specific situations, these issues may become very important. A stopping criteria should not be based on incoherent results, for instance. Also, a user creating and debugging a parallel program may wish to enforce repeatability so the exact same program sequence occurs on every run.

In the BLACS, coherence and repeatability apply only in the context of the combine operations. As mentioned above, it is possible to have communication which is incoherent (for instance, two machines which store floating point numbers differently may easily produce incoherent communication, since a number stored on machine A may not have a representation on machine B). However, the BLACS cannot control this issue. Communication is assumed to be coherent, which for communication implies that it is also repeatable.

For combine operations, the BLACS allow you to set flags indicating that you would like combines to be repeatable and/or heterogeneous coherent (see `blacs_get` and `blacs_set` for details on setting these flags).

If the BLACS are instructed to guarantee heterogeneous coherency, the BLACS restrict the topologies which can be used so that one process calculates the final result of the combine, and if necessary, broadcasts the answer to all other processes.

If the BLACS are instructed to guarantee repeatability, orderings will be enforced in the topologies which are selected. This may result in loss of performance which can range from negligible to serious depending on the application.

A couple of additional notes are in order. Incoherence and nonrepeatability can arise as a result of floating point errors, as discussed previously. This might lead you to suspect that integer calculations are always repeatable and coherent, since they involve exact arithmetic. This is true if overflow is ignored. With overflow taken into consideration, even integer calculations can display incoherence and non-repeatability. Therefore, if the repeatability or coherence flags are set, the BLACS treats integer combines the same as floating point combines in enforcing repeatability and coherence guards.

By their nature, maximization and minimization should always be repeatable. In the complex precisions, however, the real and imaginary parts must be combined in order to obtain a magnitude value used to do the comparison (this is typically $|r| + |i|$ or $\text{sqr}(r^2 + i^2)$). This allows for the possibility of heterogeneous incoherence. The BLACS therefore restrict which topologies are used for maximization and minimization in the complex routines when the heterogeneous coherence flag is set.

BLACS Combine Operations

This topic describes BLACS routines that combine the data to produce a result.

In a combine operation, each participating process contributes data that is combined with other processes' data to produce a result. This result can be given to a particular process (called the *destination* process), or to all participating processes. If the result is given to only one process, the operation is referred to as a *leave-on-one* combine, and if the result is given to all participating processes the operation is referenced as a *leave-on-all* combine.

At present, three kinds of combines are supported. They are:

- element-wise summation
- element-wise absolute value maximization
- element-wise absolute value minimization

of general rectangular arrays.

Note that a combine operation combines data between processes. By definition, a combine performed across a scope of only one process does not change the input data. This is why the operations (`max/min/sum`) are specified as *element-wise*. Element-wise indicates that each element of the input array will be combined with the corresponding element from all other processes' arrays to produce the result. Thus, a 4 x 2 array of inputs produces a 4 x 2 answer array.

When the `max/min` comparison is being performed, absolute value is used. For example, -5 and 5 are equivalent. However, the returned value is unchanged; that is, it is not the absolute value, but is a signed value instead. Therefore, if you performed a BLACS absolute value maximum combine on the numbers -5, 3, 1, 8 the result would be -8.

The initial symbol ? in the routine names below masks the data type:

i	integer
s	single precision real
d	double precision real
c	single precision complex
z	double precision complex.

BLACS Combines

Routine name	Results of operation
gamx2d	Entries of result matrix will have the value of the greatest absolute value found in that position.
gamn2d	Entries of result matrix will have the value of the smallest absolute value found in that position.
gsum2d	Entries of result matrix will have the summation of that position.

?gamx2d

Performs element-wise absolute value maximization.

Syntax

```
call igamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call sgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call dgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call cgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call zgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
<i>top</i>	CHARACTER*1. Communication pattern to use during the combine operation.
<i>m</i>	INTEGER. The number of matrix rows to be combined.
<i>n</i>	INTEGER. The number of matrix columns to be combined.

<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Matrix to be compared with to produce the maximum.
<i>lda</i>	INTEGER. The leading dimension of the matrix <i>A</i> , that is, the distance between two successive elements in a matrix row.
<i>rcflag</i>	INTEGER. If <i>rcflag</i> = -1, the arrays <i>ra</i> and <i>ca</i> are not referenced and need not exist. Otherwise, <i>rcflag</i> indicates the leading dimension of these arrays, and so must be $\geq m$.
<i>rdest</i>	INTEGER. The process row coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.
<i>cdest</i>	INTEGER. The process column coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.

Output Parameters

<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.
<i>ra</i>	INTEGER array (<i>rcflag</i> , <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> \times <i>n</i>) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.
<i>ca</i>	INTEGER array (<i>rcflag</i> , <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> \times <i>n</i>) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

Description

This routine performs element-wise absolute value maximization, that is, each element of matrix *A* is compared with the corresponding element of the other process's matrices. Note that the value of *A* is returned, but the absolute value is used to determine the maximum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

[Examples of BLACS Routines Usage](#)

?gamn2d

Performs element-wise absolute value minimization.

Syntax

```

call igamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call sgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call dgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call cgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call zgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )

```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
<i>top</i>	CHARACTER*1. Communication pattern to use during the combine operation.
<i>m</i>	INTEGER. The number of matrix rows to be combined.
<i>n</i>	INTEGER. The number of matrix columns to be combined.
<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Matrix to be compared with to produce the minimum.
<i>lda</i>	INTEGER. The leading dimension of the matrix <i>A</i> , that is, the distance between two successive elements in a matrix row.
<i>rcflag</i>	INTEGER. If <i>rcflag</i> = -1, the arrays <i>ra</i> and <i>ca</i> are not referenced and need not exist. Otherwise, <i>rcflag</i> indicates the leading dimension of these arrays, and so must be $\geq m$.
<i>rdest</i>	INTEGER. The process row coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.
<i>cdest</i>	INTEGER. The process column coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.

Output Parameters

<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.
<i>ra</i>	INTEGER array (<i>rcflag</i> , <i>n</i>).

If *rcflag* = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least *rcflag* × *n*) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

ca

INTEGER array (*rcflag*, *n*).

If *rcflag* = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least *rcflag* × *n*) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

Description

This routine performs element-wise absolute value minimization, that is, each element of matrix *A* is compared with the corresponding element of the other process's matrices. Note that the value of *A* is returned, but the absolute value is used to determine the minimum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

[Examples of BLACS Routines Usage](#)

?gsum2d

Performs element-wise summation.

Syntax

```
call igsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call sgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call dgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call cgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call zgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
<i>top</i>	CHARACTER*1. Communication pattern to use during the combine operation.
<i>m</i>	INTEGER. The number of matrix rows to be combined.
<i>n</i>	INTEGER. The number of matrix columns to be combined.
<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Matrix to be added to produce the sum.
<i>lda</i>	INTEGER. The leading dimension of the matrix <i>A</i> , that is, the distance between two successive elements in a matrix row.

rdest

INTEGER.

The process row coordinate of the process that should receive the result. If *rdest* or *cdest* = -1, all processes within the indicated scope receive the answer.

cdest

INTEGER.

The process column coordinate of the process that should receive the result. If *rdest* or *cdest* = -1, all processes within the indicated scope receive the answer.

Output Parameters

a

TYPE array (*lda*, *n*). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.

Description

This routine performs element-wise summation, that is, each element of matrix *A* is summed with the corresponding element of the other process's matrices. Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

[Examples of BLACS Routines Usage](#)

BLACS Point To Point Communication

This topic describes BLACS routines for point to point communication.

Point to point communication requires two complementary operations. The *send* operation produces a message that is then consumed by the *receive* operation. These operations have various resources associated with them. The main such resource is the buffer that holds the data to be sent or serves as the area where the incoming data is to be received. The level of *blocking* indicates what correlation the return from a send/receive operation has with the availability of these resources and with the status of message.

Non-blocking

The return from the *send* or *receive* operations does not imply that the resources may be reused, that the message has been sent/received or that the complementary operation has been called. Return means only that the send/receive has been started, and will be completed at some later date. Polling is required to determine when the operation has finished.

In non-blocking message passing, the concept of *communication/computation overlap* (abbreviated C/C overlap) is important. If a system possesses C/C overlap, independent computation can occur at the same time as communication. That means a nonblocking operation can be posted, and unrelated work can be done while the message is sent/received in parallel. If C/C overlap is not present, after returning from the routine call, computation will be interrupted at some later date when the message is actually sent or received.

Locally-blocking

Return from the *send* or *receive* operations indicates that the resources may be reused. However, since this only depends on local information, it is unknown whether the complementary operation has been called. There are no locally-blocking receives: the send must be completed before the receive buffer is available for re-use.

If a receive has not been posted at the time a locally-blocking send is issued, buffering will be required to avoid losing the message. Buffering can be done on the sending process, the receiving process, or not done at all, losing the message.

Globally-blocking

Return from a globally-blocking procedure indicates that the operation resources may be reused, and that complement of the operation has at least been posted. Since the receive has been posted, there is no buffering required for globally-blocking sends: the message is always sent directly into the user's receive buffer.

Almost all processors support non-blocking communication, as well as some other level of blocking sends. What level of blocking the send possesses varies between platforms. For instance, the Intel® processors support locally-blocking sends, with buffering done on the receiving process. This is a very important distinction, because codes written assuming locally-blocking sends will hang on platforms with globally-blocking sends. Below is a simple example of how this can occur:

```
IAM = MY_PROCESS_ID()
IF (IAM .EQ. 0) THEN
  SEND TO PROCESS 1
  RECV FROM PROCESS 1
ELSE IF (IAM .EQ. 1) THEN
  SEND TO PROCESS 0
  RECV FROM PROCESS 0
END IF
```

If the send is globally-blocking, process 0 enters the send, and waits for process 1 to start its receive before continuing. In the meantime, process 1 starts to send to 0, and waits for 0 to receive before continuing. Both processes are now waiting on each other, and the program will never continue.

The solution for this case is obvious. One of the processes simply reverses the order of its communication calls and the hang is avoided. However, when the communication is not just between two processes, but rather involves a hierarchy of processes, determining how to avoid this kind of difficulty can become problematic.

For this reason, it was decided the BLACS would support locally-blocking sends. On systems natively supporting globally-blocking sends, non-blocking sends coupled with buffering is used to simulate locally-blocking sends. The BLACS support globally-blocking receives.

In addition, the BLACS specify that point to point messages between two given processes will be strictly ordered. If process 0 sends three messages (label them *A*, *B*, and *C*) to process 1, process 1 must receive *A* before it can receive *B*, and message *C* can be received only after both *A* and *B*. The main reason for this restriction is that it allows for the computation of message identifiers.

Note, however, that messages from different processes are not ordered. If processes 0, . . . , 3 send messages *A*, . . . , *D* to process 4, process 4 may receive these messages in any order that is convenient.

Convention

The convention used in the communication routine names follows the template `?xxyy2d`, where the letter in the `?` position indicates the data type being sent, `xx` is replaced to indicate the shape of the matrix, and the `yy` positions are used to indicate the type of communication to perform:

i	integer
s	single precision real
d	double precision real
c	single precision complex

z	double precision complex
ge	The data to be communicated is stored in a general rectangular matrix.
tr	The data to be communicated is stored in a trapezoidal matrix.
sd	Send. One process sends to another.
rv	Receive. One process receives from another.

BLACS Point To Point Communication

Routine name	Operation performed
gesd2d	Take the indicated matrix and send it to the destination process.
trsd2d	
gerv2d	Receive a message from the process into the matrix.
trrv2d	

As a simple example, the pseudo code given above is rewritten below in terms of the BLACS. It is further specified that the data being exchanged is the double precision vector X , which is 5 elements long.

```
CALL GRIDINFO(NPROW, NPCOL, MYPROW, MYPCOL)

IF (MYPROW.EQ.0 .AND. MYPCOL.EQ.0) THEN
  CALL DGESD2D(5, 1, X, 5, 1, 0)
  CALL DGERV2D(5, 1, X, 5, 1, 0)
ELSE IF (MYPROW.EQ.1 .AND. MYPCOL.EQ.0) THEN
  CALL DGESD2D(5, 1, X, 5, 0, 0)
  CALL DGERV2D(5, 1, X, 5, 0, 0)
END IF
```

[?gesd2d](#)

Takes a general rectangular matrix and sends it to the destination process.

Syntax

```
call igesd2d( icontxt, m, n, a, lda, rdest, cdest )
call sgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call dgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call cgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call zgesd2d( icontxt, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rdest</i>	INTEGER. The process row coordinate of the process to send the message to.
<i>cdest</i>	INTEGER. The process column coordinate of the process to send the message to.

Description

This routine takes the indicated general rectangular matrix and sends it to the destination process located at {RDEST, CDEST} in the process grid. Return from the routine indicates that the buffer (the matrix A) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

See Also

[Examples of BLACS Routines Usage](#)

?trsd2d

Takes a trapezoidal matrix and sends it to the destination process.

Syntax

```
call itrdsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call strdsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call dtrdsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ctrdsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ztrdsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>uplo, diag, m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rdest</i>	INTEGER. The process row coordinate of the process to send the message to.
<i>cdest</i>	INTEGER. The process column coordinate of the process to send the message to.

Description

This routine takes the indicated trapezoidal matrix and sends it to the destination process located at {RDEST, CDEST} in the process grid. Return from the routine indicates that the buffer (the matrix A) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

?gerv2d

Receives a message from the process into the general rectangular matrix.

Syntax

```
call igerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call sgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call dgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call cgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call zgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<code>icontxt</code>	INTEGER. Integer handle that indicates the context.
<code>m, n, lda</code>	Describe the matrix to be sent. See Matrix Shapes for details.
<code>rsrc</code>	INTEGER. The process row coordinate of the source of the message.
<code>csrc</code>	INTEGER. The process column coordinate of the source of the message.

Output Parameters

<code>a</code>	An array of dimension (lda, n) to receive the incoming message into.
----------------	--

Description

This routine receives a message from process {RSRC, CSRC} into the general rectangular matrix *A*. This routine is globally-blocking, that is, return from the routine indicates that the message has been received into *A*.

See Also

[Examples of BLACS Routines Usage](#)

?trrv2d

Receives a message from the process into the trapezoidal matrix.

Syntax

```
call itrsv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call strsv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call dtrsv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call ctrsv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call ztrsv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<code>icontxt</code>	INTEGER. Integer handle that indicates the context.
<code>uplo, diag, m, n, lda</code>	Describe the matrix to be sent. See Matrix Shapes for details.
<code>rsrc</code>	INTEGER. The process row coordinate of the source of the message.
<code>csrc</code>	INTEGER. The process column coordinate of the source of the message.

Output Parameters

<code>a</code>	An array of dimension (lda, n) to receive the incoming message into.
----------------	--

Description

This routine receives a message from process {RSRC, CSRC} into the trapezoidal matrix A. This routine is globally-blocking, that is, return from the routine indicates that the message has been received into A.

BLACS Broadcast Routines

This topic describes BLACS broadcast routines.

A broadcast sends data possessed by one process to all processes within a scope. Broadcast, much like point to point communication, has two complementary operations. The process that owns the data to be broadcast issues a *broadcast/send*. All processes within the same scope must then issue the complementary *broadcast/receive*.

The BLACS define that both broadcast/send and broadcast/receive are *globally-blocking*. Broadcasts/receives cannot be locally-blocking since they must post a receive. Note that receives cannot be locally-blocking. When a given process can leave, a broadcast/receive operation is topology dependent, so, to avoid a hang as topology is varied, the broadcast/receive must be treated as if no process can leave until all processes have called the operation.

Broadcast/sends could be defined to be *locally-blocking*. Since no information is being received, as long as locally-blocking point to point sends are used, the broadcast/send will be locally blocking. However, defining one process within a scope to be locally-blocking while all other processes are globally-blocking adds little to the programmability of the code. On the other hand, leaving the option open to have globally-blocking broadcast/sends may allow for optimization on some platforms.

The fact that broadcasts are defined as globally-blocking has several important implications. The first is that scoped operations (broadcasts or combines) must be strictly ordered, that is, all processes within a scope must agree on the order of calls to separate scoped operations. This constraint falls in line with that already in place for the computation of message IDs, and is present in point to point communication as well.

A less obvious result is that scoped operations with `SCOPE = 'ALL'` must be ordered with respect to any other scoped operation. This means that if there are two broadcasts to be done, one along a column, and one involving the entire process grid, all processes within the process column issuing the column broadcast must agree on which broadcast will be performed first.

The convention used in the communication routine names follows the template `?xxyy2d`, where the letter in the ? position indicates the data type being sent, `xx` is replaced to indicate the shape of the matrix, and the `yy` positions are used to indicate the type of communication to perform:

i	integer
s	single precision real
d	double precision real
c	single precision complex
z	double precision complex
ge	The data to be communicated is stored in a general rectangular matrix.
tr	The data to be communicated is stored in a trapezoidal matrix.
bs	Broadcast/send. A process begins the broadcast of data within a scope.
br	Broadcast/receive A process receives and participates in the broadcast of data within a scope.

BLACS Broadcast Routines

Routine name	Operation performed
<code>gebs2d</code>	Start a broadcast along a scope.

Routine name	Operation performed
trbs2d	
gebr2d	Receive and participate in a broadcast along a scope.
trbr2d	
Product and Performance Information	
Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex .	
Notice revision #20201201	

?gebs2d

Starts a broadcast along a scope for a general rectangular matrix.

Syntax

```
call igebs2d( icontxt, scope, top, m, n, a, lda )
call sgebs2d( icontxt, scope, top, m, n, a, lda )
call dgebs2d( icontxt, scope, top, m, n, a, lda )
call cgebs2d( icontxt, scope, top, m, n, a, lda )
call zgebs2d( icontxt, scope, top, m, n, a, lda )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix *A*.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

See Also

[Examples of BLACS Routines Usage](#)

?trbs2d

Starts a broadcast along a scope for a trapezoidal matrix.

Syntax

```
call itrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
```



```
call strbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call dtrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call ctrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call ztrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>uplo, diag, m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix A.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

?gebr2d

Receives and participates in a broadcast along a scope for a general rectangular matrix.

Syntax

```
call igebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call sgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call dgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call cgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call zgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER. The process row coordinate of the process that called broadcast/send.

csrc

INTEGER.

The process column coordinate of the process that called broadcast/send.

Output Parameters

a

An array of dimension (lda, n) to receive the incoming message into.

Description

This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix *A*. Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

See Also

[Examples of BLACS Routines Usage](#)

?trbr2d

Receives and participates in a broadcast along a scope for a trapezoidal matrix.

Syntax

```
call itrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call strbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call dtrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call ctrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call ztrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
```

Input Parameters

icontxt

INTEGER. Integer handle that indicates the context.

scope

CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.

top

CHARACTER*1. Indicates the communication pattern to use for the broadcast.

uplo, diag, m, n, lda

Describe the matrix to be sent. See [Matrix Shapes](#) for details.

rsrc

INTEGER.

The process row coordinate of the process that called broadcast/send.

csrc

INTEGER.

The process column coordinate of the process that called broadcast/send.

Output Parameters

a

An array of dimension (lda, n) to receive the incoming message into.

Description

This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix *A*. Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

BLACS Support Routines

The support routines perform distinct tasks that can be used for:

Initialization

Destruction

Information Purposes

Miscellaneous Tasks.

Initialization Routines

This topic describes BLACS routines that deal with grid/context creation, and processing before the grid/context has been defined.

BLACS Initialization Routines

Routine name	Operation performed
blacs_pinfo	Returns the number of processes available for use.
blacs_setup	Allocates virtual machine and spawns processes.
blacs_get	Gets values that BLACS use for internal defaults.
blacs_set	Sets values that BLACS use for internal defaults.
blacs_gridinit	Assigns available processes into BLACS process grid.
blacs_gridmap	Maps available processes into BLACS process grid.

[blacs_pinfo](#)

Returns the number of processes available for use.

Syntax

```
call blacs_pinfo( mypnum, nprocs )
```

Output Parameters

<i>mypnum</i>	INTEGER. An integer between 0 and (<i>nprocs</i> - 1) that uniquely identifies each process.
<i>nprocs</i>	INTEGER. The number of processes available for BLACS use.

Description

This routine is used when some initial system information is required before the BLACS are set up. On all platforms except PVM, *nprocs* is the actual number of processes available for use, that is, *nprows* * *npcols* <= *nprocs*. In PVM, the virtual machine may not have been set up before this call, and therefore no parallel machine exists. In this case, *nprocs* is returned as less than one. If a process has been spawned via the

keyboard, it receives *mypnum* of 0, and all other processes get *mypnum* of -1. As a result, the user can distinguish between processes. Only after the virtual machine has been set up via a call to `BLACS_SETUP`, this routine returns the correct values for *mypnum* and *nprocs*.

See Also

[Examples of BLACS Routines Usage](#)

blacs_setup

Allocates virtual machine and spawns processes.

Syntax

```
call blacs_setup( mypnum, nprocs )
```

Input Parameters

<i>nprocs</i>	INTEGER. On the process spawned from the keyboard rather than from <code>pvmspawn</code> , this parameter indicates the number of processes to create when building the virtual machine.
---------------	--

Output Parameters

<i>mypnum</i>	INTEGER. An integer between 0 and (<i>nprocs</i> - 1) that uniquely identifies each process.
<i>nprocs</i>	INTEGER. For all processes other than spawned from the keyboard, this parameter means the number of processes available for BLACS use.

Description

This routine only accomplishes meaningful work in the PVM BLACS. On all other platforms, it is functionally equivalent to `blacs_pinfo`. The BLACS assume a static system, that is, the given number of processes does not change. PVM supplies a dynamic system, allowing processes to be added to the system on the fly.

`blacs_setup` is used to allocate the virtual machine and spawn off processes. It reads in a file called `blacs_setup.dat`, in which the first line must be the name of your executable. The second line is optional, but if it exists, it should be a PVM spawn flag. Legal values at this time are 0 (`PvmTaskDefault`), 4 (`PvmTaskDebug`), 8 (`PvmTaskTrace`), and 12 (`PvmTaskDebug + PvmTaskTrace`). The primary reason for this line is to allow the user to easily turn on and off PVM debugging. Additional lines, if any, specify what machines should be added to the current configuration before spawning *nprocs*-1 processes to the machines in a round robin fashion.

nprocs is input on the process which has no PVM parent (that is, *mypnum*=0), and both parameters are output for all processes. So, on PVM systems, the call to `blacs_pinfo` informs you that the virtual machine has not been set up, and a call to `blacs_setup` then sets up the machine and returns the real values for *mypnum* and *nprocs*.

Note that if the file `blacs_setup.dat` does not exist, the BLACS prompt the user for the executable name, and processes are spawned to the current PVM configuration.

See Also

[Examples of BLACS Routines Usage](#)

blacs_get

Gets values that BLACS use for internal defaults.

Syntax

```
call blacs_get( ictxt, what, val )
```

Input Parameters

<i>ictxt</i>	INTEGER. On values of <i>what</i> that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.
<i>what</i>	<p>INTEGER. Indicates what BLACS internal(s) should be returned in <i>val</i>. Present options are:</p> <ul style="list-style-type: none"> • <i>what</i> = 0 : Handle indicating default system context. • <i>what</i> = 1 : The BLACS message ID range. • <i>what</i> = 2 : The BLACS debug level the library was compiled with. • <i>what</i> = 10 : Handle indicating the system context used to define the BLACS context whose handle is <i>ictxt</i>. • <i>what</i> = 11 : Number of rings multiring broadcast topology is presently using. • <i>what</i> = 12 : Number of branches general tree broadcast topology is presently using. • <i>what</i> = 13 : Number of rings multiring combine topology is presently using. • <i>what</i> = 14 : Number of branches general tree combine topology is presently using. • <i>what</i> = 15 : Whether topologies are forced to be repeatable or not. A non-zero return value indicates that topologies are being forced to be repeatable. See Repeatability and Coherence for more information about repeatability. • <i>what</i> = 16 : Whether topologies are forced to be heterogenous coherent or not. A non-zero return value indicates that topologies are being forced to be heterogenous coherent. See Repeatability and Coherence for more information about coherence.

Output Parameters

<i>val</i>	INTEGER. The value of the BLACS internal.
------------	---

Description

This routine gets the values that the BLACS are using for internal defaults. Some values are tied to a BLACS context, and some are more general. The most common use is in retrieving a default system context for input into [blacs_gridinit](#) or [blacs_gridmap](#).

Some systems, such as MPI*, supply their own version of context. For those users who mix system code with BLACS code, a BLACS context should be formed in reference to a system context. Thus, the grid creation routines take a system context as input. If you wish to have strictly portable code, you may use [blacs_get](#) to retrieve a default system context that will include all available processes. This value is not tied to a BLACS context, so the parameter *ictxt* is unused.

[blacs_get](#) returns information on three quantities that are tied to an individual BLACS context, which is passed in as *ictxt*. The information that may be retrieved is:

- The handle of the system context upon which this BLACS context was defined
- The number of rings for TOP = 'M' (multiring broadcast/combine)

- The number of branches for `TOP = 'T'` (general tree broadcast/general tree gather).
- Whether topologies are being forced to be repeatable or heterogenous coherent.

See Also

Examples of BLACS Routines Usage

blacs_set

Sets values that BLACS use for internal defaults.

Syntax

```
call blacs_set( ictxt, what, val )
```

Input Parameters

<i>ictxt</i>	INTEGER. For values of <i>what</i> that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.
<i>what</i>	<p>INTEGER. Indicates what BLACS internal(s) should be set. Present values are:</p> <ul style="list-style-type: none"> • 1 = Set the BLACS message ID range • 11 = Number of rings for multiring broadcast topology to use • 12 = Number of branches for general tree broadcast topology to use • 13 = Number of rings for multiring combine topology to use • 14 = Number of branches for general tree combine topology to use • 15 = Force topologies to be repeatable or not • 16 = Force topologies to be heterogenous coherent or not
<i>val</i>	INTEGER. Array of dimension (*). Indicates the value(s) the internals should be set to. The specific meanings depend on <i>what</i> values, as discussed in Description below.

Description

This routine sets the BLACS internal defaults depending on *what* values:

<i>what</i> = 1	<p>Setting the BLACS message ID range.</p> <p>If you wish to mix the BLACS with other message-passing packages, restrict the BLACS to a certain message ID range not to be used by the non-BLACS routines. The message ID range must be set before the first call to <code>blacs_gridinit</code> or <code>blacs_gridmap</code>. Subsequent calls will have no effect. Because the message ID range is not tied to a particular context, the parameter <i>ictxt</i> is ignored, and <i>val</i> is defined as:</p> <p>VAL (input) INTEGER array of dimension (2)</p> <p>VAL(1) : The smallest message ID (also called message type or message tag) the BLACS should use.</p> <p>VAL(2) : The largest message ID (also called message type or message tag) the BLACS should use.</p>
<i>what</i> = 11	<p>Set number of rings for <code>TOP = 'M'</code> (multiring broadcast). This quantity is tied to a context, so <i>ictxt</i> is used, and <i>val</i> is defined as:</p>

	VAL (input) INTEGER array of dimension (1)
	VAL(1) : The number of rings for multiring topology to use.
<i>what</i> = 12	Set number of branches for TOP = 'T' (general tree broadcast). This quantity is tied to a context, so <i>icontxt</i> is used, and <i>val</i> is defined as:
	VAL (input) INTEGER array of dimension (1)
	VAL(1) : The number of branches for general tree topology to use.
<i>what</i> = 13	Set number of rings for TOP = 'M' (multiring combine). This quantity is tied to a context, so <i>icontxt</i> is used, and <i>val</i> is defined as:
	VAL (input) INTEGER array of dimension (1)
	VAL(1) : The number of rings for multiring topology to use.
<i>what</i> = 14	Set number of branches for TOP = 'T' (general tree gather). This quantity is tied to a context, so <i>icontxt</i> is used, and <i>val</i> is defined as:
	VAL (input) INTEGER array of dimension (1)
	VAL(1) : The number of branches for general tree topology to use.
<i>what</i> = 15	Force topologies to be repeatable or not (see Repeatability and Coherence for more information about repeatability).
	VAL (input) INTEGER array of dimension (1)
	VAL(1) = 0 (default) Topologies are not required to be repeatable.
	VAL(1) ≠ 0 All used topologies are required to be repeatable, which might degrade performance.
<i>what</i> = 16	Force topologies to be heterogenous coherent or not (see Repeatability and Coherence for more information about coherence).
	VAL (input) INTEGER array of dimension (1)
	VAL(1) = 0 (default) Topologies are not required to be heterogenous coherent.
	VAL(1) ≠ 0 All used topologies are required to be heterogenous coherent, which might degrade performance.

blacs_gridinit

Assigns available processes into BLACS process grid.

Syntax

call blacs_gridinit(*icontxt*, *layout*, *nprow*, *npcol*)

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call <code>blacs_get</code> to obtain a default system context.
<i>layout</i>	CHARACTER*1. Indicates how to map processes to BLACS grid. Options are:

- 'R' : Use row-major natural ordering
- 'C' : Use column-major natural ordering
- ELSE : Use row-major natural ordering

nprow

INTEGER. Indicates how many process rows the process grid should contain.

npcol

INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

icontxt

INTEGER. Integer handle to the created BLACS context.

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridmap`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine creates a simple `nprow` x `npcol` process grid. This process grid uses the first `nprow * npcol` processes, and assigns them to the grid in a row- or column-major natural ordering. If these process-to-grid mappings are unacceptable, call `blacs_gridmap`.

See Also

Examples of BLACS Routines Usage

`blacs_get``blacs_gridmap``blacs_setup`

`blacs_gridmap`

Maps available processes into BLACS process grid.

Syntax

```
call blacs_gridmap( icontxt, usermap, ldumap, nprow, npcol )
```

Input Parameters

icontxt

INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call `blacs_get` to obtain a default system context.

<i>usermap</i>	INTEGER. Array, dimension (<i>ldumap</i> , <i>npcol</i>), indicating the process-to-grid mapping.
<i>ldumap</i>	INTEGER. Leading dimension of the 2D array <i>usermap</i> . <i>ldumap</i> \geq <i>nprow</i> .
<i>nprow</i>	INTEGER. Indicates how many process rows the process grid should contain.
<i>npcol</i>	INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

<i>icontxt</i>	INTEGER. Integer handle to the created BLACS context.
----------------	---

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridinit`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are actual processors (hardware), and they are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine allows the user to map processes to the process grid in an arbitrary manner. `usermap(i,j)` holds the process number of the process to be placed in $\{i, j\}$ of the process grid. On most distributed systems, this process number is a machine defined number between $0 \dots nprow-1$. For PVM, these node numbers are the PVM TIDS (Task IDs). The `blacs_gridmap` routine is intended for an experienced user. The `blacs_gridinit` routine is much simpler. `blacs_gridinit` simply performs a `gridmap` where the first $nprow * npcold$ processes are mapped into the current grid in a row-major natural ordering. If you are an experienced user, `blacs_gridmap` allows you to take advantage of your system's actual layout. That is, you can map nodes that are physically connected to be neighbors in the BLACS grid, etc. The `blacs_gridmap` routine also opens the way for *multigridding*: you can separate your nodes into arbitrary grids, join them together at some later date, and then re-split them into new grids. `blacs_gridmap` also provides the ability to make arbitrary grids or subgrids (for example, a "nearest neighbor" grid), which can greatly facilitate operations among processes that do not fall on a row or column of the main process grid.

See Also

Examples of BLACS Routines Usage

`blacs_get`

`blacs_gridinit`

`blacs_setup`

Destruction Routines

This topic describes BLACS routines that destroy grids, abort processes, and free resources.

BLACS Destruction Routines

Routine name	Operation performed
blacs_freebuff	Frees BLACS buffer.
blacs_gridexit	Frees a BLACS context.
blacs_abort	Aborts all processes.
blacs_exit	Frees all BLACS contexts and releases all allocated memory.

[blacs_freebuff](#)

Frees BLACS buffer.

Syntax

```
call blacs_freebuff( ictxt, wait )
```

Input Parameters

<i>ictxt</i>	INTEGER. Integer handle that indicates the BLACS context.
<i>wait</i>	INTEGER. Parameter indicating whether to wait for non-blocking operations or not. If equals 0, the operations should not be waited for; free only unused buffers. Otherwise, wait in order to free all buffers.

Description

This routine releases the BLACS buffer.

The BLACS have at least one internal buffer that is used for packing messages. The number of internal buffers depends on what platform you are running the BLACS on. On systems where memory is tight, keeping this buffer or buffers may become expensive. Call `freebuff` to release the buffer. However, the next call of a communication routine that requires packing reallocates the buffer.

The *wait* parameter determines whether the BLACS should wait for any non-blocking operations to be completed or not. If *wait* = 0, the BLACS free any buffers that can be freed without waiting. If *wait* is not 0, the BLACS free all internal buffers, even if non-blocking operations must be completed first.

[blacs_gridexit](#)

Frees a BLACS context.

Syntax

```
call blacs_gridexit( ictxt )
```

Input Parameters

<i>ictxt</i>	INTEGER. Integer handle that indicates the BLACS context to be freed.
--------------	---

Description

This routine frees a BLACS context.

Release the resources when contexts are no longer needed. After freeing a context, the context no longer exists, and its handle may be re-used if new contexts are defined.

blacs_abort

Aborts all processes.

Syntax

```
call blacs_abort( icontxt, errornum )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the BLACS context to be aborted.
<i>errornum</i>	INTEGER. User-defined integer error number.

Description

This routine aborts all the BLACS processes, not only those confined to a particular context.

Use `blacs_abort` to abort all the processes in case of a serious error. Note that both parameters are input, but the routine uses them only in printing out the error message. The context handle passed in is not required to be a valid context handle.

blacs_exit

Frees all BLACS contexts and releases all allocated memory.

Syntax

```
call blacs_exit( continue )
```

Input Parameters

<i>continue</i>	INTEGER. Flag indicating whether message passing continues after the BLACS are done. If <i>continue</i> is non-zero, the user is assumed to continue using the machine after completing the BLACS. Otherwise, no message passing is assumed after calling this routine.
-----------------	---

Description

This routine frees all BLACS contexts and releases all allocated memory.

This routine should be called when a process has finished all use of the BLACS. The *continue* parameter indicates whether the user will be using the underlying communication platform after the BLACS are finished. This information is most important for the PVM BLACS. If *continue* is set to 0, then `pvm_exit` is called; otherwise, it is not called. Setting *continue* not equal to 0 indicates that explicit PVM `send/recvs` will be called after the BLACS routines are used. Make sure your code calls `pvm_exit`. PVM users should either call `blacs_exit` or explicitly call `pvm_exit` to avoid PVM problems.

See Also

[Examples of BLACS Routines Usage](#)

Informational Routines

This topic describes BLACS routines that return information involving the process grid.

BLACS Informational Routines

Routine name	Operation performed
blacs_gridinfo	Returns information on the current grid.
blacs_pnum	Returns the system process number of the process in the process grid.
blacs_pcoord	Returns the row and column coordinates in the process grid.

[blacs_gridinfo](#)

Returns information on the current grid.

Syntax

```
call blacs_gridinfo( ictxt, nprow, npc, myprow, mypcol )
```

Input Parameters

ictxt INTEGER. Integer handle that indicates the context.

Output Parameters

nprow INTEGER. Number of process rows in the current process grid.

npcol INTEGER. Number of process columns in the current process grid.

myprow INTEGER. Row coordinate of the calling process in the process grid.

mypcol INTEGER. Column coordinate of the calling process in the process grid.

Description

This routine returns information on the current grid. If the context handle does not point at a valid context, all quantities are returned as -1.

See Also

[Examples of BLACS Routines Usage](#)

[blacs_pnum](#)

Returns the system process number of the process in the process grid.

Syntax

```
call blacs_pnum( ictxt, prow, pcol )
```

Input Parameters

ictxt INTEGER. Integer handle that indicates the context.

prow INTEGER. Row coordinate of the process the system process number of which is to be determined.

pcol INTEGER. Column coordinate of the process the system process number of which is to be determined.

Description

This function returns the system process number of the process at {PROW, PCOL} in the process grid.

See Also

[Examples of BLACS Routines Usage](#)

blacs_pcoord

Returns the row and column coordinates in the process grid.

Syntax

```
call blacs_pcoord( icontxt, pnum, prow, pcol )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>pnum</i>	INTEGER. Process number the coordinates of which are to be determined. This parameter stand for the process number of the underlying machine, that is, it is a <code>tid</code> for PVM.

Output Parameters

<i>prow</i>	INTEGER. Row coordinates of the <i>pnum</i> process in the BLACS grid.
<i>pcol</i>	INTEGER. Column coordinates of the <i>pnum</i> process in the BLACS grid.

Description

Given the system process number, this function returns the row and column coordinates in the BLACS process grid.

See Also

[Examples of BLACS Routines Usage](#)

Miscellaneous Routines

This topic describes `blacs_barrier` routine.

BLACS Informational Routines

Routine name	Operation performed
blacs_barrier	Holds up execution of all processes within the indicated scope until they have all called the routine.

blacs_barrier

Holds up execution of all processes within the indicated scope.

Syntax

```
call blacs_barrier( icontxt, scope )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
----------------	---

scope

CHARACTER*1. Parameter that indicates whether a process row (*scope*='R'), column ('C'), or entire grid ('A') will participate in the barrier.

Description

This routine holds up execution of all processes within the indicated scope until they have all called the routine.

Examples of BLACS Routines Usage

Data Fitting Functions

Data Fitting functions in Intel® oneAPI Math Kernel Library (oneMKL) provide spline-based interpolation capabilities that you can use to approximate functions, function derivatives or integrals, and perform cell search operations.

The Data Fitting component is task based. The task is a data structure or descriptor that holds the parameters related to a specific Data Fitting operation. You can modify the task parameters using the task editing functionality of the library.

For definition of the implemented operations, see [Mathematical Conventions](#).

Data Fitting routines use the following workflow to process a task:

1. Create a task or multiple tasks.
2. Modify the task parameters.
3. Perform a Data Fitting computation.
4. Destroy the task or tasks.

All Data Fitting functions fall into the following categories:

[Task Creation and Initialization Routines](#) - routines that create a new Data Fitting task descriptor and initialize the most common parameters, such as partition of the interpolation interval, values of the vector-valued function, and the parameters describing their structure.

[Task Configuration Routines](#) - routines that set, modify, or query parameters in an existing Data Fitting task.

[Computational Routines](#) - routines that perform Data Fitting computations, such as construction of a spline, interpolation, computation of derivatives and integrals, and search.

[Task Destructors](#) - routines that delete Data Fitting task descriptors and deallocate resources.

You can access the Data Fitting routines through the Fortran and C89/C99 language interfaces. You can also use the C89 interface with more recent versions of C/C++, or the Fortran 90 interface with programs written in Fortran 95.

The `$(MKL)/include` directory of the Intel® oneAPI Math Kernel Library (oneMKL) contains the following Data Fitting header files:

- `mkl_df.h`

You can find examples that demonstrate usage of Data Fitting routines in the `$(MKL)/examples/datafittingc` directory .

Data Fitting Function Naming Conventions

The interface of the Data Fitting functions, types, and constants are case-sensitive and can be in lowercase, uppercase, and mixed case.

The names of all routines have the following structure:

`df[datatype]<base_name>`

where

- `df` is a prefix indicating that the routine belongs to the Data Fitting component of Intel® oneAPI Math Kernel Library (oneMKL).
- `[datatype]` field specifies the type of the input and/or output data and can be `s` (for the single precision real type), `d` (for the double precision real type), or `i` (for the integer type). This field is omitted in the names of the routines that are not data type dependent.
- `<base_name>` field specifies the functionality the routine performs. For example, this field can be `NewTask1D`, `Interpolate1D`, or `DeleteTask`

Data Fitting Function Data Types

The Data Fitting component provides routines for processing single and double precision real data types. The results of cell search operations are returned as a generic integer data type.

All Data Fitting routines use the following data type:

Type	Data Object
<code>DFTaskPtr</code>	Pointer to a task

NOTE

The actual size of the generic integer type is platform-dependent. Before compiling your application, you need to set an appropriate byte size for integers. For details, see section *Using the ILP64 Interface vs. LP64 Interface of the Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*.

Mathematical Conventions for Data Fitting Functions

This section explains the notation used for Data Fitting function descriptions. Spline notations are based on the terminology and definitions of [deBoor2001]. The Subbotin quadratic spline definition follows the conventions of [StechSub76]. The quasi-uniform partition definition is based on [Schumaker2007].

Mathematical Notation in the Data Fitting Component

Concept	Mathematical Notation
Partition of interpolation interval $[a, b]$, where <ul style="list-style-type: none"> • x_i denotes breakpoints. • $[x_i, x_{i+1})$ denotes a sub-interval (cell) of size $\Delta_i = x_{i+1} - x_i$. 	$\{x_i\}_{i=1,\dots,n}$, where $a = x_1 < x_2 < \dots < x_n = b$
Quasi-uniform partition of interpolation interval $[a, b]$	Partition $\{x_i\}_{i=1,\dots,n}$ which meets the constraint with a constant C defined as $1 \leq M/m \leq C,$ where <ul style="list-style-type: none"> • $M = \max_{i=1,\dots,n-1} (\Delta_i)$ • $m = \min_{i=1,\dots,n-1} (\Delta_i)$ • $\Delta_i = x_{i+1} - x_i$
Vector-valued function of dimension p being fit	$f(x) = (f_1(x), \dots, f_p(x))$
Piecewise polynomial (PP) function f of order $k+1$	$f(x) := P_i(x)$, if $x \in [x_i, x_{i+1})$, $i = 1, \dots, n-1$ where

Concept	Mathematical Notation
Function p agrees with function f at the points $\{x_i\}_{i=1,\dots,n}$.	<ul style="list-style-type: none"> $\{x_i\}_{i=1,\dots,n}$ is a strictly increasing sequence of breakpoints. $P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + \dots + c_{i,k}(x - x_i)^k$ is a polynomial of degree k (order $k+1$) over the interval $x \in [x_i, x_{i+1})$.
The k -th divided difference of function f at points x_i, \dots, x_{i+k} . This difference is the leading coefficient of the polynomial of order $k+1$ that agrees with f at x_i, \dots, x_{i+k} .	<p>For every point ζ in sequence $\{x_i\}_{i=1,\dots,n}$ that occurs m times, the equality $p^{(i-1)}(\zeta) = f^{(i-1)}(\zeta)$ holds for all $i = 1, \dots, m$, where $p^{(i)}(t)$ is the derivative of the i-th order.</p> <p>$[x_i, \dots, x_{i+k}] f$</p> <p>In particular,</p> <ul style="list-style-type: none"> $[x_1] f = f(x_1)$ $[x_1, x_2] f = (f(x_1) - f(x_2)) / (x_1 - x_2)$
A k -order derivative of interpolant $f(x)$ at interpolation site τ .	$f^{(k)}(\tau)$

Interpolants to the Function f at x_1, \dots, x_n and Boundary Conditions

Concept	Mathematical Notation
Linear interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i),$ where <ul style="list-style-type: none"> $x \in [x_i, x_{i+1})$ $c_{1,i} = f(x_i)$ $c_{2,i} = [x_i, x_{i+1}] f$ $i = 1, \dots, n-1$
Piecewise parabolic interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2, x \in [x_i, x_{i+1})$ Coefficients $c_{1,i}$, $c_{2,i}$, and $c_{3,i}$ depend on the conditions: <ul style="list-style-type: none"> $P_i(x_i) = f(x_i)$ $P_i(x_{i+1}) = f(x_{i+1})$ $P_i((x_{i+1} + x_i) / 2) = v_{i+1}$ where parameter v_{i+1} depends on the interpolant being continuously differentiable: $P_{i-1}^{(1)}(x_i) = P_i^{(1)}(x_i)$
Piecewise parabolic Subbotin interpolant	$P(x) = P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + d_{3,i}((x - t_i)_+)^2,$ where <ul style="list-style-type: none"> $x \in [t_i, t_{i+1})$ $\{t_i\}_{i=1,\dots,n+1}$ is a sequence of knots such that <ul style="list-style-type: none"> $t_1 = x_1, t_{n+1} = x_n$ $t_i \in (x_{i-1}, x_i), i = 2, \dots, n$ $x_+ = f(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$ <p>Coefficients $c_{1,i}$, $c_{2,i}$, $c_{3,i}$, and $d_{3,i}$ depend on the following conditions:</p>

Concept	Mathematical Notation
	<ul style="list-style-type: none"> $P_i(x_i) = f(x_i), P_i(x_{i+1}) = f(x_{i+1})$ $P(x)$ is a continuously differentiable polynomial of the second degree on $[t_i, t_{i+1}), i = 1, \dots, n$.
Piecewise cubic Hermite interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$ <p>where</p> <ul style="list-style-type: none"> $x \in [x_i, x_{i+1})$ $c_{1,i} = f(x_i)$ $c_{2,i} = s_i$ $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ $i = 1, \dots, n-1$ $s_i = f^{(1)}(x_i)$
Piecewise cubic Bessel interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$ <p>where</p> <ul style="list-style-type: none"> $x \in [x_i, x_{i+1})$ $c_{1,i} = f(x_i)$ $c_{2,i} = s_i$ $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ $i = 1, \dots, n-1$ $s_i = (\Delta x_i[x_{i-1}, x_i]f + \Delta x_{i-1}[x_i, x_{i+1}]f) / (\Delta x_i + \Delta x_{i+1})$
Piecewise cubic Akima interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$ <p>where</p> <ul style="list-style-type: none"> $x \in [x_i, x_{i+1})$ $c_{1,i} = f(x_i)$ $c_{2,i} = s_i$ $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ $i = 1, \dots, n-1$ $s_i = (w_{i+1}[x_{i-1}, x_i]f + w_{i-1}[x_i, x_{i+1}]f) / (w_{i+1} + w_{i-1}),$ <p>where</p> $w_i = [x_i, x_{i+1}]f - [x_{i-1}, x_i]f $
Piecewise natural cubic interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$ <p>where</p> <ul style="list-style-type: none"> $x \in [x_i, x_{i+1})$ $c_{1,i} = f(x_i)$ $c_{2,i} = s_i$ $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ $i = 1, \dots, n-1$ Parameter s_i depends on the condition that the interpolant is twice continuously differentiable: $P_{i-1}^{(2)}(x_i) = P_i^{(2)}(x_i)$.
Not-a-knot boundary condition.	Parameters s_1 and s_n provide $P_1 = P_2$ and $P_{n-1} = P_n$, so that the first and the last interior breakpoints are inactive.
Free-end boundary condition.	$f''(x_1) = f''(x_n) = 0$

Concept	Mathematical Notation
Look-up interpolator for discrete set of points $(x_1, y_1), \dots, (x_n, y_n)$.	$y(x) = \begin{cases} y_1, & \text{if } x = x_1 \\ y_2, & \text{if } x = x_2 \\ \dots & \\ y_n, & \text{if } x = x_n \\ \text{error,} & \text{otherwise} \end{cases}$
Step-wise constant continuous right interpolator.	$y(x) = \begin{cases} y_1, & \text{if } x_1 \leq x < x_2 \\ y_2, & \text{if } x_2 \leq x < x_3 \\ \dots & \\ y_{n-1}, & \text{if } x_{n-1} \leq x < x_n \\ y_n, & \text{if } x = x_n \end{cases}$
Step-wise constant continuous left interpolator.	$y(x) = \begin{cases} y_1, & \text{if } x = x_1 \\ y_2, & \text{if } x_1 < x \leq x_2 \\ y_3, & \text{if } x_2 < x \leq x_3 \\ \dots & \\ y_n, & \text{if } x_{n-1} < x \leq x_n \end{cases}$

Data Fitting Usage Model

Consider an algorithm that uses the Data Fitting functions. Typically, such algorithms consist of four steps or stages:

1. Create a task. You can call the Data Fitting function several times to create multiple tasks.

```
status = dfdNewTask1D( &task, nx, x, xhint, ny, y, yhint );
```

2. Modify the task parameters.

```
status = dfdEditPPSpline1D( task, s_order, c_type, bc_type, bc, ic_type, ic,
scoeff, scoeffhint );
```

3. Perform Data Fitting spline-based computations. You may reiterate steps 2-3 as needed.

```
status = dfdInterpolate1D(task, estimate, method, nsite, site, sitehint, ndorder,
dorder, datahint, r, rhint, cell );
```

4. Destroy the task or tasks.

```
status = dfDeleteTask( &task );
```

See Also

[Data Fitting Usage Examples](#)

Data Fitting Usage Examples

The examples below illustrate several operations that you can perform with Data Fitting routines.

You can get source code for similar examples in the `.\examples\datafittingc` subdirectory of the Intel® oneAPI Math Kernel Library (oneMKL) installation directory.

The following example demonstrates the construction of a linear spline using Data Fitting routines. The spline approximates a scalar function defined on non-uniform partition. The coefficients of the spline are returned as a one-dimensional array:

Example of Linear Spline Construction

```
#include "mkl.h"
#define N 500 /* Size of partition, number of breakpoints */
#define SPLINE_ORDER DF_PP_LINEAR /* Linear spline to construct */

int main()
{
    int status; /* Status of a Data Fitting operation */
    DFTaskPtr task; /* Data Fitting operations are task based */

    /* Parameters describing the partition */
    MKL_INT nx; /* The size of partition x */
    double x[N]; /* Partition x */
    MKL_INT xhint; /* Additional information about the structure of breakpoints */

    /* Parameters describing the function */
    MKL_INT ny; /* Function dimension */
    double y[N]; /* Function values at the breakpoints */
    MKL_INT yhint; /* Additional information about the function */

    /* Parameters describing the spline */
    MKL_INT s_order; /* Spline order */
    MKL_INT s_type; /* Spline type */
    MKL_INT ic_type; /* Type of internal conditions */
    double* ic; /* Array of internal conditions */
    MKL_INT bc_type; /* Type of boundary conditions */
    double* bc; /* Array of boundary conditions */

    double scoeff[(N-1)* SPLINE_ORDER]; /* Array of spline coefficients */
    MKL_INT scoeffhint; /* Additional information about the coefficients */

    /* Initialize the partition */
    nx = N;
    /* Set values of partition x */
    ...
    xhint = DF_NO_HINT; /* No additional information about the function is provided.
                        By default, the partition is non-uniform. */
    /* Initialize the function */
    ny = 1; /* The function is scalar. */

    /* Set function values */
    ...
    yhint = DF_NO_HINT; /* No additional information about the function is provided. */

    /* Create a Data Fitting task */
    status = dfdNewTask1D( &task, nx, x, xhint, ny, y, yhint );

    /* Check the Data Fitting operation status */
    ...

    /* Initialize spline parameters */
    s_order = DF_PP_LINEAR; /* Spline is of the second order. */
    s_type = DF_PP_DEFAULT; /* Spline is of the default type. */
}
```

```

/* Define internal conditions for linear spline construction (none in this example) */
ic_type = DF_NO_IC;
ic = NULL;

/* Define boundary conditions for linear spline construction (none in this example) */
bc_type = DF_NO_BC;
bc = NULL;
scoeffhint = DF_NO_HINT;    /* No additional information about the spline. */

/* Set spline parameters in the Data Fitting task */
status = dfdEditPPSpline1D( task, s_order, s_type, bc_type, bc, ic_type,
                           ic, scoeff, scoeffhint );

/* Check the Data Fitting operation status */
...

/* Use a standard computation method to construct a linear spline: */
/*  $P_i(x) = c_{i,0} + c_{i,1}(x - x_i)$ ,  $i=0, \dots, N-2$  */
/* The library packs spline coefficients to array scoeff. */
/*  $scoeff[2*i+0] = c_{i,0}$  and  $scoeff[2*i+1] = c_{i,1}$ ,  $i=0, \dots, N-2$  */
status = dfdConstruct1D( task, DF_PP_SPLINE, DF_METHOD_STD );

/* Check the Data Fitting operation status */
...

/* Process spline coefficients */
...

/* Deallocate Data Fitting task resources */
status = dfDeleteTask( &task );

/* Check the Data Fitting operation status */
...
return 0 ;
}

```

The following example demonstrates cubic spline-based interpolation using Data Fitting routines. In this example, a scalar function defined on non-uniform partition is approximated by Bessel cubic spline using not-a-knot boundary conditions. Once the spline is constructed, you can use the spline to compute spline values at the given sites. Computation results are packed by the Data Fitting routine in row-major format.

Example of Cubic Spline-Based Interpolation

```

#include "mkl.h"

#define NX 100                /* Size of partition, number of breakpoints */
#define NSITE 1000           /* Number of interpolation sites */
#define SPLINE_ORDER DF_PP_CUBIC /* A cubic spline to construct */

int main()
{
    int status;                /* Status of a Data Fitting operation */
    DFTaskPtr task;            /* Data Fitting operations are task based */

    /* Parameters describing the partition */
    MKL_INT nx;                /* The size of partition x */
    double x[NX];              /* Partition x */
    MKL_INT xhint;             /* Additional information about the structure of breakpoints */

```

```

/* Parameters describing the function */
MKL_INT ny;          /* Function dimension */
double y[NX];        /* Function values at the breakpoints */
MKL_INT yhint;       /* Additional information about the function */

/* Parameters describing the spline */
MKL_INT s_order;     /* Spline order */
MKL_INT s_type;      /* Spline type */
MKL_INT ic_type;     /* Type of internal conditions */
double* ic;          /* Array of internal conditions */
MKL_INT bc_type;     /* Type of boundary conditions */
double* bc;          /* Array of boundary conditions */

double scoeff[(NX-1)* SPLINE_ORDER]; /* Array of spline coefficients */
MKL_INT scoeffhint; /* Additional information about the coefficients */

/* Parameters describing interpolation computations */
MKL_INT nsite;       /* Number of interpolation sites */
double site[NSITE]; /* Array of interpolation sites */
MKL_INT sitehint;    /* Additional information about the structure of
                      interpolation sites */

MKL_INT ndorder, dorder; /* Parameters defining the type of interpolation */

double* datahint; /* Additional information on partition and interpolation sites */

double r[NSITE]; /* Array of interpolation results */
MKL_INT rhint; /* Additional information on the structure of the results */
MKL_INT* cell; /* Array of cell indices */

/* Initialize the partition */
nx = NX;

/* Set values of partition x */
...
xhint = DF_NON_UNIFORM_PARTITION; /* The partition is non-uniform. */

/* Initialize the function */
ny = 1; /* The function is scalar. */

/* Set function values */
...
yhint = DF_NO_HINT; /* No additional information about the function is provided. */

/* Create a Data Fitting task */
status = dfdNewTask1D( &task, nx, x, xhint, ny, y, yhint );

/* Check the Data Fitting operation status */
...

/* Initialize spline parameters */
s_order = DF_PP_CUBIC; /* Spline is of the fourth order (cubic spline). */
s_type = DF_PP_BESSEL; /* Spline is of the Bessel cubic type. */

/* Define internal conditions for cubic spline construction (none in this example) */
ic_type = DF_NO_IC;
ic = NULL;

```

```

/* Use not-a-knot boundary conditions. In this case, the is first and the last
   interior breakpoints are inactive, no additional values are provided. */
bc_type = DF_BC_NOT_A_KNOT;
bc = NULL;
scoeffhint = DF_NO_HINT; /* No additional information about the spline. */

/* Set spline parameters in the Data Fitting task */
status = dfdEditPPSpline1D( task, s_order, s_type, bc_type, bc, ic_type,
                           ic, scoeff, scoeffhint );

/* Check the Data Fitting operation status */
...

/* Use a standard method to construct a cubic Bessel spline: */
/*  $P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + c_{i,2}(x - x_i)^2 + c_{i,3}(x - x_i)^3$ , */
/* The library packs spline coefficients to array scoeff: */
/* scoeff[4*i+0] = ci,0, scoeff[4*i+1] = ci,1, */
/* scoeff[4*i+2] = ci,2, scoeff[4*i+3] = ci,3, */
/* i=0,...,N-2 */
status = dfdConstruct1D( task, DF_PP_SPLINE, DF_METHOD_STD );

/* Check the Data Fitting operation status */
...

/* Initialize interpolation parameters */
nsite = NSITE;

/* Set site values */
...

sitehint = DF_NON_UNIFORM_PARTITION; /* Partition of sites is non-uniform */

/* Request to compute spline values */
ndorder = 1;
dorder = 1;
datahint = DF_NO_APRIORI_INFO; /* No additional information about breakpoints or
                               sites is provided. */
rhint = DF_MATRIX_STORAGE_ROWS; /* The library packs interpolation results
                                in row-major format. */
cell = NULL; /* Cell indices are not required. */

/* Solve interpolation problem using the default method: compute the spline values
   at the points site(i), i=0,..., nsite-1 and place the results to array r */
status = dfdInterpolate1D( task, DF_INTERP, DF_METHOD_PP, nsite, site,
                          sitehint, ndorder, &dorder, datahint, r, rhint, cell );

/* Check Data Fitting operation status */
...

/* De-allocate Data Fitting task resources */
status = dfDeleteTask( &task );
/* Check Data Fitting operation status */
...
return 0;
}

```

The following example demonstrates how to compute indices of cells containing given sites. This example uses uniform partition presented with two boundary points. The sites are in the ascending order.

Example of Cell Search

```
#include "mkl.h"

#define NX 100                      /* Size of partition, number of breakpoints */
#define NSITE 1000                  /* Number of interpolation sites */

int main()
{
    int status;                    /* Status of a Data Fitting operation */
    DFTaskPtr task;                /* Data Fitting operations are task based */

    /* Parameters describing the partition */
    MKL_INT nx;                   /* The size of partition x */
    float x[2];                   /* Partition x is uniform and holds endpoints
                                   of interpolation interval [a, b] */
    MKL_INT xhint;                /* Additional information about the structure of breakpoints */

    /* Parameters describing the function */
    MKL_INT ny;                   /* Function dimension */
    float *y;                     /* Function values at the breakpoints */
    MKL_INT yhint;                /* Additional information about the function */

    /* Parameters describing cell search */
    MKL_INT nsite;                /* Number of interpolation sites */
    float site[NSITE];            /* Array of interpolation sites */
    MKL_INT sitehint;             /* Additional information about the structure of sites */

    float* datahint;              /* Additional information on partition and interpolation sites */

    MKL_INT cell[NSITE];          /* Array for cell indices */

    /* Initialize a uniform partition */
    nx = NX;
    /* Set values of partition x: for uniform partition, */
    /* provide end-points of the interpolation interval [-1.0,1.0] */
    x[0] = -1.0f; x[1] = 1.0f;
    xhint = DF_UNIFORM_PARTITION; /* Partition is uniform */

    /* Initialize function parameters */
    /* In cell search, function values are not necessary and are set to zero/NULL values */
    ny = 0;
    y = NULL;
    yhint = DF_NO_HINT;

    /* Create a Data Fitting task */
    status = dfsNewTask1D( &task, nx, x, xhint, ny, y, yhint );

    /* Check Data Fitting operation status */
    ...

    /* Initialize interpolation (cell search) parameters */
    nsite = NSITE;

    /* Set sites in the ascending order */
    ...
    sitehint = DF_SORTED_DATA;      /* Sites are provided in the ascending order. */
    datahint = DF_NO_APRIORI_INFO; /* No additional information
```

```

        about breakpoints/sites is provided.*/

/* Use a standard method to compute indices of the cells that contain
   interpolation sites. The library places the index of the cell containing
   site(i) to the cell(i), i=0,...,nsite-1 */
status = dfsSearchCells1D( task, DF_METHOD_STD, nsite, site, sitehint,
                          datahint, cell );
/* Check Data Fitting operation status */
...

/* Process cell indices */
...

/* Deallocate Data Fitting task resources */
status = dfDeleteTask( &task );

/* Check Data Fitting operation status */
...
return 0;
}

```

Data Fitting Function Task Status and Error Reporting

The Data Fitting routines report a task status through integer values. Negative status values indicate errors, while positive values indicate warnings. An error can be caused by invalid parameter values or a memory allocation failure.

The status codes have symbolic names predefined in the header file as macros via the `#define` statements.

If no error occurred, the function returns the `DF_STATUS_OK` code defined as zero:

```
#define DF_STATUS_OK 0
```

In case of an error, the function returns a non-zero error code that specifies the origin of the failure. Header files define the following status codes:

Status Codes in the Data Fitting Component

Status Code	Description
Common Status Codes	
<code>DF_STATUS_OK</code>	Operation completed successfully.
<code>DF_ERROR_NULL_TASK</code>	Data Fitting task is a <code>NULL</code> pointer.
<code>DF_ERROR_MEM_FAILURE</code>	Memory allocation failure.
<code>DF_ERROR_METHOD_NOT_SUPPORTED</code>	Requested method is not supported.
<code>DF_ERROR_COMP_TYPE_NOT_SUPPORTED</code>	Requested computation type is not supported.
<code>DF_ERROR_NULL_PTR</code>	Pointer to parameter is null.
Data Fitting Task Creation and Initialization, and Generic Editing Operations	
<code>DF_ERROR_BAD_NX</code>	Invalid number of breakpoints.
<code>DF_ERROR_BAD_X</code>	Array of breakpoints is invalid.
<code>DF_ERROR_BAD_X_HINT</code>	Invalid hint describing the structure of the partition.

Status Code	Description
DF_ERROR_BAD_NY	Invalid dimension of vector-valued function y .
DF_ERROR_BAD_Y	Array of function values is invalid.
DF_ERROR_BAD_Y_HINT	Invalid flag describing the structure of function y
Data Fitting Task-Specific Editing Operations	
DF_ERROR_BAD_SPLINE_ORDER	Invalid spline order.
DF_ERROR_BAD_SPLINE_TYPE	Invalid spline type.
DF_ERROR_BAD_IC_TYPE	Type of internal conditions used for spline construction is invalid.
DF_ERROR_BAD_IC	Array of internal conditions for spline construction is not defined.
DF_ERROR_BAD_BC_TYPE	Type of boundary conditions used in spline construction is invalid.
DF_ERROR_BAD_BC	Array of boundary conditions for spline construction is not defined.
DF_ERROR_BAD_PP_COEFF	Array of piecewise polynomial spline coefficients is not defined.
DF_ERROR_BAD_PP_COEFF_HINT	Invalid flag describing the structure of the piecewise polynomial spline coefficients.
DF_ERROR_BAD_PERIODIC_VAL	Function values at the endpoints of the interpolation interval are not equal as required in periodic boundary conditions.
DF_ERROR_BAD_DATA_ATTR	Invalid attribute of the pointer to be set or modified in Data Fitting task descriptor with the <code>df?</code> <code>EditIdxPtr</code> task editor.
DF_ERROR_BAD_DATA_IDX	Index of the pointer to be set or modified in the Data Fitting task descriptor with the <code>df?</code> <code>EditIdxPtr</code> task editor is out of the pre-defined range.
Data Fitting Computation Operations	
DF_ERROR_BAD_NSITE	Invalid number of interpolation sites.
DF_ERROR_BAD_SITE	Array of interpolation sites is not defined.
DF_ERROR_BAD_SITE_HINT	Invalid flag describing the structure of interpolation sites.
DF_ERROR_BAD_NDORDER	Invalid size of the array defining derivative orders to be computed at interpolation sites.
DF_ERROR_BAD_DORDER	Array defining derivative orders to be computed at interpolation sites is not defined.
DF_ERROR_BAD_DATA_HINT	Invalid flag providing additional information about partition or interpolation sites.

Status Code	Description
DF_ERROR_BAD_INTERP	Array of spline-based interpolation results is not defined.
DF_ERROR_BAD_INTERP_HINT	Invalid flag defining the structure of spline-based interpolation results.
DF_ERROR_BAD_CELL_IDX	Array of indices of partition cells containing interpolation sites is not defined.
DF_ERROR_BAD_NLIM	Invalid size of arrays containing integration limits.
DF_ERROR_BAD_LIM	Array of the left-side integration limits is not defined.
DF_ERROR_BAD_RLIM	Array of the right-side integration limits is not defined.
DF_ERROR_BAD_INTEGR	Array of spline-based integration results is not defined.
DF_ERROR_BAD_INTEGR_HINT	Invalid flag providing the structure of the array of spline-based integration results.
DF_ERROR_BAD_LOOKUP_INTERP_SITE	Bad site provided for interpolation with look-up interpolator.

NOTE

The routine that estimates piecewise polynomial cubic spline coefficients can return internal error codes related to the specifics of the implementation. Such error codes indicate invalid input data or other issues unrelated to Data Fitting routines.

Data Fitting Task Creation and Initialization Routines

Task creation and initialization routines are functions used to create a new task descriptor and initialize its parameters. The Data Fitting component provides the `df?NewTask1D` routine that creates and initializes a new task descriptor for a one-dimensional Data Fitting task.

`df?NewTask1D`

Creates and initializes a new task descriptor for a one-dimensional Data Fitting task.

Syntax

```
status = dfsNewTask1D(&task, nx, x, xhint, ny, y, yhint)
```

```
status = dfdNewTask1D(&task, nx, x, xhint, ny, y, yhint)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>nx</i>	const MKL_INT	Number of breakpoints representing partition of interpolation interval $[a, b]$.
<i>x</i>	const float* for dfsNewTask1D const double* for dfdNewTask1D	One-dimensional array containing the strictly sorted breakpoints from interpolation interval $[a, b]$. The structure of the array is defined by parameter <i>xhint</i> : <ul style="list-style-type: none"> If partition is non-uniform or quasi-uniform, the array should contain <i>nx</i> strictly ordered values. If partition is uniform, the array should contain two entries that represent endpoints of interpolation interval $[a, b]$. <hr/> Caution The array must be strictly sorted. If it is unordered, the results of data fitting routines are not correct. <hr/>
<i>xhint</i>	const MKL_INT	A flag describing the structure of partition <i>x</i> . For the list of possible values of <i>xhint</i> , see table "Hint Values for Partition x" . If you set the flag to the DF_NO_HINT value, the library interprets the partition as non-uniform.
<i>ny</i>	const MKL_INT	Dimension of vector-valued function <i>y</i> .
<i>y</i>	const float* for dfsNewTask const double* for dfdNewTask	Vector-valued function <i>y</i> , array of size <i>nx</i> * <i>ny</i> . The storage format of function values in the array is defined by the value of flag <i>yhint</i> .
<i>yhint</i>	const MKL_INT	A flag describing the structure of array <i>y</i> . Valid hint values are listed in table "Hint Values for Vector-Valued Function y" . If you set the flag to the DF_NO_HINT value, the library assumes that all <i>ny</i> coordinates of the vector-valued function <i>y</i> are provided and stored in row-major format.

Output Parameters

Name	Type	Description
<i>task</i>	DFTaskPtr	Descriptor of the task.
<i>status</i>	int	Status of the routine: <ul style="list-style-type: none"> DF_STATUS_OK if the task is created successfully. Non-zero error code if the task creation failed. See "Task Status and Error Reporting" for error code definitions.

Description

The `df?NewTask1D` routine creates and initializes a new Data Fitting task descriptor with user-specified parameters for a one-dimensional Data Fitting task. The x and nx parameters representing the partition of interpolation interval $[a, b]$ are mandatory. If you provide invalid values for these parameters, such as a `NULL` pointer x or the number of breakpoints smaller than two, the routine does not create the Data Fitting task and returns an error code.

If you provide a vector-valued function y , make sure that the function dimension ny and the array of function values y are both valid. If any of these parameters are invalid, the routine does not create the Data Fitting task and returns an error code.

If you store coordinates of the vector-valued function y in non-contiguous memory locations, you can set the `yhint` flag to `DF_1ST_COORDINATE`, and pass only the first coordinate of the function into the task creation routine. After successful creation of the Data Fitting task, you can pass the remaining coordinates using the `df?EditIdxPtr` task editor.

If the routine fails to create the task descriptor, it returns a `NULL` task pointer.

The routine supports the following hint values for partition x :

Hint Values for Partition x

Value	Description
<code>DF_NON_UNIFORM_PARTITION</code>	Partition is non-uniform.
<code>DF_QUASI_UNIFORM_PARTITION</code>	Partition is quasi-uniform.
<code>DF_UNIFORM_PARTITION</code>	Partition is uniform.
<code>DF_NO_HINT</code>	No hint is provided. By default, partition is interpreted as non-uniform.

The routine supports the following hint values for the vector-valued function:

Hint Values for Vector-Valued Function y

Value	Description
<code>DF_MATRIX_STORAGE_ROWS</code>	Data is stored in row-major format according to C conventions.
<code>DF_MATRIX_STORAGE_COLS</code>	Data is stored in column-major format according to Fortran conventions.
<code>DF_1ST_COORDINATE</code>	The first coordinate of vector-valued data is provided.
<code>DF_NO_HINT</code>	No hint is provided. By default, the coordinates of vector-valued function y are provided and stored in row-major format.

NOTE

You must preserve the arrays x (breakpoints) and y (vector-valued functions) through the entire workflow of the Data Fitting computations for a task, as the task stores the addresses of the arrays for spline-based computations.

Task Configuration Routines

In order to configure tasks, you can use task editors and task query routines.

Task editors initialize or change the predefined Data Fitting task parameters. You can use task editors to initialize or modify pointers to arrays or parameter values.

Task editors can be task-specific or generic. Task-specific editors can modify more than one parameter related to a specific task. Generic editors modify a single parameter at a time.

The Data Fitting component of Intel® oneAPI Math Kernel Library (oneMKL) provides the following task editors:

Data Fitting Task Editors

Editor	Description	Type
<code>df?</code> <code>EditPPSpline1D</code>	Changes parameters of the piecewise polynomial spline.	Task-specific
<code>df?EditPtr</code>	Changes a pointer in the task descriptor.	Generic
<code>dfiEditVal</code>	Changes a value in the task descriptor.	Generic
<code>df?EditIdxPtr</code>	Changes a coordinate of data represented in matrix format, such as a vector-valued function or spline coefficients.	Generic

Task query routines are used to read the predefined Data Fitting task parameters. You can use task query routines to read the values of pointers or parameters.

Task query routines are generic (not task-specific), allowing you to read a single parameter at a time.

The Data Fitting component of the Intel® oneAPI Math Kernel Library (oneMKL) provides the following task query routines:

Data Fitting Task Query Routines

Editor	Description	Type
<code>df?QueryPtr</code>	Queries a pointer in the task descriptor.	Generic
<code>dfiQueryVal</code>	Queries a value in the task descriptor.	Generic
<code>df?QueryIdxPtr</code>	Queries a coordinate of data represented in matrix format, such as a vector-valued function or spline coefficients.	Generic

`df?EditPPSpline1D`

Modifies parameters representing a spline in a Data Fitting task descriptor.

Syntax

```
status = dfsEditPPSpline1D(task, s_order, s_type, bc_type, bc, ic_type, ic, scoeff,
scoeffhint)
```

```
status = dfdEditPPSpline1D(task, s_order, s_type, bc_type, bc, ic_type, ic, scoeff,
scoeffhint)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	DFTaskPtr	Descriptor of the task.
<i>s_order</i>	const MKL_INT	Spline order. The parameter takes one of the values described in table "Spline Orders Supported by Data Fitting Functions" .
<i>s_type</i>	const MKL_INT	Spline type. The parameter takes one of the values described in table "Spline Types Supported by Data Fitting Functions" .
<i>bc_type</i>	const MKL_INT	Type of boundary conditions. The parameter takes one of the values described in table "Boundary Conditions Supported by Data Fitting Functions" .
<i>bc</i>	const float* for dfsEditPPSpline1D const double* for dfdEditPPSpline1D	Pointer to boundary conditions. The size of the array is defined by the value of parameter <i>bc_type</i> : <ul style="list-style-type: none"> • If you set free-end or not-a-knot boundary conditions, pass the NULL pointer to this parameter. • If you combine boundary conditions at the endpoints of the interpolation interval, pass an array of two elements. • If you set a boundary condition for the default quadratic spline or a periodic condition for Hermite or the default cubic spline, pass an array of one element.
<i>ic_type</i>	const MKL_INT	Type of internal conditions. The parameter takes one of the values described in table "Internal Conditions Supported by Data Fitting Functions" .
<i>ic</i>	const float* for dfsEditPPSpline1D const double* for dfdEditPPSpline1D	A non-NULL pointer to the array of internal conditions. The size of the array is defined by the value of parameter <i>ic_type</i> : <ul style="list-style-type: none"> • If you set first derivatives or second derivatives internal conditions (<i>ic_type</i>=DF_IC_1ST_DER or <i>ic_type</i>=DF_IC_2ND_DER), pass an array of <i>n</i>-1 derivative values at the internal points of the interpolation interval. • If you set the knot values internal condition for Subbotin spline (<i>ic_type</i>=DF_IC_Q_KNOT) and the knot partition is non-uniform, pass an array of <i>n</i>+1 elements. • If you set the knot values internal condition for Subbotin spline (<i>ic_type</i>=DF_IC_Q_KNOT) and the knot partition is uniform, pass an array of four elements.
<i>scoeff</i>	const float* for dfsEditPPSpline1D const double* for dfdEditPPSpline1D	Spline coefficients. An array of size <i>ny*s_order*(nx-1)</i> . The storage format of the coefficients in the array is defined by the value of flag <i>scoeffhint</i> .

Name	Type	Description
<code>scoeffhint</code>	<code>const MKL_INT</code>	A flag describing the structure of the array of spline coefficients. For valid hint values, see table "Hint Values for Spline Coefficients" . The library stores the coefficients in row-major format. The default value is <code>DF_NO_HINT</code> .

Output Parameters

Name	Type	Description
<code>status</code>	<code>int</code>	<p>Status of the routine:</p> <ul style="list-style-type: none"> <code>DF_STATUS_OK</code> if the routine execution completed successfully. Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.

Description

The editor modifies parameters that describe the order, type, boundary conditions, internal conditions, and coefficients of a spline. The spline order definition is provided in the ["Mathematical Conventions"](#) section. You can set the spline order to any value supported by Data Fitting functions. The table below lists the available values:

Spline Orders Supported by the Data Fitting Functions

Order	Description
<code>DF_PP_STD</code>	Artificial value. Use this value for look-up and step-wise constant interpolants only.
<code>DF_PP_LINEAR</code>	Piecewise polynomial spline of the second order (linear spline).
<code>DF_PP_QUADRATIC</code>	Piecewise polynomial spline of the third order (quadratic spline).
<code>DF_PP_CUBIC</code>	Piecewise polynomial spline of the fourth order (cubic spline).

To perform computations with a spline not supported by Data Fitting routines, set the parameter defining the spline order and pass the spline coefficients to the library in the supported format. For format description, see figure ["Row-major Coefficient Storage Format"](#).

The table below lists the supported spline types:

Spline Types Supported by Data Fitting Functions

Type	Description
<code>DF_PP_DEFAULT</code>	The default spline type. You can use this type with linear, quadratic, or user-defined splines.
<code>DF_PP_SUBBOTIN</code>	Quadratic splines based on Subbotin algorithm, [TechSub76] .
<code>DF_PP_NATURAL</code>	Natural cubic spline.

Type	Description
DF_PP_HERMITE	Hermite cubic spline.
DF_PP_BESSEL	Bessel cubic spline.
DF_PP_AKIMA	Akima cubic spline.
DF_LOOKUP_INTERPOLANT	Look-up interpolant.
DF_CR_STEPWISE_CONST_INTERPOLANT	Continuous right step-wise constant interpolant.
DF_CL_STEPWISE_CONST_INTERPOLANT	Continuous left step-wise constant interpolant.

If you perform computations with look-up or step-wise constant interpolants, set the spline order to the `DF_PP_STD` value.

Construction of specific splines may require boundary or internal conditions. To compute coefficients of such splines, you should pass boundary or internal conditions to the library by specifying the type of the conditions and providing the necessary values. For splines that do not require additional conditions, such as linear splines, set condition types to `DF_NO_BC` and `DF_NO_IC`, and pass `NULL` pointers to the conditions. The table below defines the supported boundary conditions:

Boundary Conditions Supported by Data Fitting Functions

Boundary Condition	Description	Spline
DF_NO_BC	No boundary conditions provided.	All
DF_BC_NOT_A_KNOT	Not-a-knot boundary conditions.	Akima, Bessel, Hermite, natural cubic
DF_BC_FREE_END	Free-end boundary conditions.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_1ST_LEFT_DER	The first derivative at the left endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_1ST_RIGHT_DER	The first derivative at the right endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_2ST_LEFT_DER	The second derivative at the left endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_2ND_RIGHT_DER	The second derivative at the right endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_PERIODIC	Periodic boundary conditions.	Linear, all cubic splines
DF_BC_Q_VAL	Function value at point $(x_0 + x_1)/2$	Default quadratic

NOTE

To construct a natural cubic spline, pass these settings to the editor:

- `DF_PP_CUBIC` as the spline order,
- `DF_PP_NATURAL` as the spline type, and
- `DF_BC_FREE_END` as the boundary condition.

To construct a cubic spline with other boundary conditions, pass these settings to the editor:

- `DF_PP_CUBIC` as the spline order,
- `DF_PP_NATURAL` as the spline type, and
- the required type of boundary condition.

For Akima, Hermite, Bessel, and default cubic splines use the corresponding type defined in [Table Spline Types Supported by Data Fitting Functions](#).

You can combine the values of boundary conditions with a bitwise `OR` operation. This permits you to pass combinations of first and second derivatives at the endpoints of the interpolation interval into the library. To pass a first derivative at the left endpoint and a second derivative at the right endpoint, set the boundary conditions to `DF_BC_1ST_LEFT_DER OR DF_BC_2ND_RIGHT_DER`.

You should pass the combined boundary conditions as an array of two elements. The first entry of the array contains the value of the boundary condition for the left endpoint of the interpolation interval, and the second entry - for the right endpoint. Pass other boundary conditions as arrays of one element.

For the conditions defined as a combination of valid values, the library applies the following rules to identify the boundary condition type:

- If not required for spline construction, the value of boundary conditions is ignored.
- Not-a-knot condition has the highest priority. If set, other boundary conditions are ignored.
- Free-end condition has the second priority after the not-a-knot condition. If set, other boundary conditions are ignored.
- Periodic boundary condition has the next priority after the free-end condition.
- The first derivative has higher priority than the second derivative at the right and left endpoints.

If you set the periodic boundary condition, make sure that function values at the endpoints of the interpolation interval are identical. Otherwise, the library returns an error code. The table below specifies the values to be provided for each type of spline if the periodic boundary condition is set.

Boundary Requirements for Periodic Conditions

Spline Type	Periodic Boundary Condition Support	Boundary Value
Linear	Yes	Not required
Default quadratic	No	
Subbotin quadratic	No	
Natural cubic	Yes	Not required
Bessel	Yes	Not required
Akima	Yes	Not required
Hermite cubic	Yes	First derivative
Default cubic	Yes	Second derivative

Internal conditions supported in the Data Fitting domain that you can use for the *ic_type* parameter are the following:

Internal Conditions Supported by Data Fitting Functions

Internal Condition	Description	Spline
DF_NO_IC	No internal conditions provided.	
DF_IC_1ST_DER	Array of first derivatives of size $n-2$, where n is the number of breakpoints. Derivatives are applicable to each coordinate of the vector-valued function.	Hermite cubic
DF_IC_2ND_DER	Array of second derivatives of size $n-2$, where n is the number of breakpoints. Derivatives are applicable to each coordinate of the vector-valued function.	Default cubic
DF_IC_Q_KNOT	Knot array of size $n+1$, where n is the number of breakpoints.	Subbotin quadratic

To construct a Subbotin quadratic spline, you have three options to get the array of knots in the library:

- If you do not provide the knots, the library uses the default values of knots $t = \{t_i\}$, $i = 0, \dots, n$ according to the rule:

$$t_0 = x_0, t_n = x_{n-1}, t_i = (x_i + x_{i-1})/2, i = 1, \dots, n-1.$$

- If you provide the knots in an array of size $n+1$, the knots form a non-uniform partition. Make sure that the knot values you provide meet the following conditions:

$$t_0 = x_0, t_n = x_{n-1}, t_i \in (x_{i-1}, x_i), i = 1, \dots, n-1.$$

- If you provide the knots in an array of size 4, the knots form a uniform partition

$$t_0 = x_0, t_1 = l, t_2 = r, t_3 = x_{n-1}, \text{ where } l \in (x_0, x_1) \text{ and } r \in (x_{n-2}, x_{n-1}).$$

In this case, you need to set the value of the *ic_type* parameter holding the type of internal conditions to DF_IC_Q_KNOT OR DF_UNIFORM_PARTITION.

NOTE

Since the partition is uniform, perform an OR operation with the DF_UNIFORM_PARTITION partition hint value described in [Table Hint Values for Partition x](#).

For computations based on look-up and step-wise constant interpolants, you can avoid calling the `df?EditPPSpline1D` editor and directly call one of the routines for spline-based computation of spline values, derivatives, or integrals. For example, you can call the `df?Construct1D` routine to construct the required spline with the given attributes, such as order or type.

The memory location of the spline coefficients is defined by the *scoeff* parameter. Make sure that the size of the array is sufficient to hold $ny * s_order * (nx-1)$ values.

The `df?EditPPSpline1D` routine supports the following hint values for spline coefficients:

Hint Values for Spline Coefficients

Order	Description
DF_1ST_COORDINATE	The first coordinate of vector-valued data is provided.
DF_NO_HINT	No hint is provided. By default, all sets of spline coefficients are stored in row-major format.

The coefficients for all coordinates of the vector-valued function are packed in memory one by one in successive order, from function y_1 to function y_{ny} .

Within each coordinate, the library stores the coefficients as an array, in row-major format:

$$c_{1,0}, c_{1,1}, \dots, c_{1,k}, c_{2,0}, c_{2,1}, \dots, c_{2,k}, \dots, c_{n-1,0}, c_{n-1,1}, \dots, c_{n-1,k}$$

Mapping of the coefficients to storage in the *scoeff* array is described below, where $c_{i,j}$ is the j th coefficient of the function

$$P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + \dots + c_{i,k}(x - x_i)^k$$

.

See [Mathematical Conventions](#) for more details on nomenclature and interpolants.

Row-major Coefficient Storage Format

$$\begin{aligned}
 P_1(x) &= \overrightarrow{c_{1,0}} + \overrightarrow{c_{1,1}(x - x_1)} + \overrightarrow{\dots} + \overrightarrow{c_{1,k}(x - x_1)^k} \\
 P_2(x) &= \overrightarrow{c_{2,0}} + \overrightarrow{c_{2,1}(x - x_2)} + \overrightarrow{\dots} + \overrightarrow{c_{2,k}(x - x_2)^k} \\
 &\quad \vdots \\
 P_{n-1}(x) &= \overrightarrow{c_{n-1,0}} + \overrightarrow{c_{n-1,1}(x - x_{n-1})} + \overrightarrow{\dots} + \overrightarrow{c_{n-1,k}(x - x_{n-1})^k}
 \end{aligned}$$

If you store splines corresponding to different coordinates of the vector-valued function at non-contiguous memory locations, do the following:

1. Set the *scoeffhint* flag to DF_1ST_COORDINATE and provide the spline for the first coordinate.
2. Pass the spline coefficients for the remaining coordinates into the Data Fitting task using the *df?EditIdxPtr* task editor.

Using the *df?EditPPSpline1D* task editor, you can provide to the Data Fitting task an already constructed spline that you want to use in computations. To ensure correct interpretation of the memory content, you should set the following parameters:

- Spline order and type, if appropriate. If the spline is not supported by the library, set the *s_type* parameter to DF_PP_DEFAULT.
- Pointer to the array of spline coefficients in row-major format.
- The *scoeffhint* parameter describing the structure of the array:
 - Set the *scoeffhint* flag to the DF_1ST_COORDINATE value to pass spline coefficients stored at different memory locations. In this case, you can set the parameters that describe boundary and internal conditions to zero.

- Use the default value `DF_NO_HINT` for all other cases.

Before passing an already constructed spline into the library, you should call the `dfiEditVal` task editor to provide the dimension of the spline `DF_NY`. See table ["Parameters Supported by the dfiEditVal Task Editor"](#) for details.

After you provide the spline to the Data Fitting task, you can run computations that use this spline.

NOTE

You must preserve the arrays *bc* (boundary conditions), *ic* (internal conditions), and *scoeff* (spline coefficients) through the entire workflow of the Data Fitting computations for a task, as the task stores the addresses of the arrays for spline-based computations.

df?EditPtr

Modifies a pointer to an array held in a Data Fitting task descriptor.

Syntax

```
status = dfsEditPtr(task, ptr_attr, ptr)
```

```
status = dfdEditPtr(task, ptr_attr, ptr)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	<code>DFTaskPtr</code>	Descriptor of the task.
<i>ptr_attr</i>	<code>const MKL_INT</code>	The parameter to change. For details, see the <i>Pointer Attribute</i> column in table "Pointers Supported by the df?EditPtr Task Editor" .
<i>ptr</i>	<code>const float*</code> for <code>dfsEditPtr</code> <code>const double*</code> for <code>dfdEditPtr</code>	New pointer. For details, see the <i>Purpose</i> column in table "Pointers Supported by the df?EditPtr Task Editor" .

Output Parameters

Name	Type	Description
<i>status</i>	<code>int</code>	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The `df?EditPtr` editor replaces the pointer of type *ptr_attr* stored in a Data Fitting task descriptor with a new pointer *ptr*. The table below describes types of pointers supported by the editor:

Pointers Supported by the `df?EditPtr` Task Editor

Pointer Attribute	Purpose
<code>DF_X</code>	Partition x of the interpolation interval, an array of strictly sorted breakpoints.
	Caution The array must be strictly sorted. If it is unordered, the results of data fitting routines are not correct.
<code>DF_Y</code>	Vector-valued function y
<code>DF_IC</code>	Internal conditions for spline construction. For details, see table "Internal Conditions Supported by Data Fitting Functions" .
<code>DF_BC</code>	Boundary conditions for spline construction. For details, see table "Boundary Conditions Supported by Data Fitting Functions" .
<code>DF_PP_SCOEFF</code>	Spline coefficients

You can use `df?EditPtr` to modify different types of pointers including pointers to the vector-valued function and spline coefficients stored in contiguous memory. Use the `df?EditIdxPtr` editor if you need to modify pointers to coordinates of the vector-valued function or spline coefficients stored at non-contiguous memory locations.

If you modify a partition of the interpolation interval, then you should call the `dfiEditVal` task editor with the corresponding value of `DF_XHINT`, even if the structure of the partition remains the same.

If you pass a `NULL` pointer to the `df?EditPtr` task editor, the task remains unchanged and the routine returns an error code. For the predefined error codes, please see ["Task Status and Error Reporting"](#).

NOTE

You must preserve the arrays x (breakpoints), y (vector-valued functions), bc (boundary conditions), ic (internal conditions), and $scoeff$ (spline coefficients) through the entire workflow of the Data Fitting computations which use those arrays, as the task stores the addresses of the arrays for spline-based computations.

`dfiEditVal`

Modifies a parameter value in a Data Fitting task descriptor.

Syntax

```
status = dfiEditVal(task, val_attr, val)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>DFTaskPtr</code>	Descriptor of the task.

Name	Type	Description
<code>val_attr</code>	<code>const MKL_INT</code>	The parameter to change. See table "Parameters Supported by the dfiEditVal Task Editor" .
<code>val</code>	<code>const MKL_INT</code>	A new parameter value. See table "Parameters Supported by the dfiEditVal Task Editor" .

Output Parameters

Name	Type	Description
<code>status</code>	<code>int</code>	<p>Status of the routine:</p> <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The `dfiEditVal` task editor replaces the parameter of type `val_attr` stored in a Data Fitting task descriptor with a new value `val`. The table below describes valid types of parameter `val_attr` supported by the editor:

Parameters Supported by the dfiEditVal Task Editor

Parameter Attribute	Purpose
<code>DF_NX</code>	Number of breakpoints
<code>DF_XHINT</code>	A flag describing the structure of partition. See table "Hint Values for Partition x" for the list of available values.
<code>DF_NY</code>	Dimension of the vector-valued function
<code>DF_YHINT</code>	A flag describing the structure of the vector-valued function. See table "Hint Values for Vector Function y" for the list of available values.
<code>DF_SPLINE_ORDER</code>	Spline order. See table "Spline Orders Supported by Data Fitting Functions" for the list of available values.
<code>DF_SPLINE_TYPE</code>	Spline type. See table "Spline Types Supported by Data Fitting Functions" for the list of available values.
<code>DF_BC_TYPE</code>	Type of boundary conditions used in spline construction. See table "Boundary Conditions Supported by Data Fitting Functions" for the list of available values.
<code>DF_IC_TYPE</code>	Type of internal conditions used in spline construction. See table "Internal Conditions Supported by Data Fitting Functions" for the list of available values.
<code>DF_PP_COEFF_HINT</code>	A flag describing the structure of spline coefficients. See table "Hint Values for Spline Coefficients" for the list of available values.
<code>DF_CHECK_FLAG</code>	A flag which controls checking of Data Fitting parameters. See table "Possible Values for the DF_CHECK_FLAG Parameter" for the list of available values.

If you pass a zero value for the parameter describing the size of the arrays that hold coefficients for a partition, a vector-valued function, or a spline, the parameter held in the Data fitting task remains unchanged and the routine returns an error code. For the predefined error codes, see ["Task Status and Error Reporting"](#).

Possible Values for the `DF_CHECK_FLAG` Parameter

Value	Description
<code>DF_ENABLE_CHECK_FLAG</code>	Checks the correctness of parameters of Data Fitting computational routines (default mode).
<code>DF_DISABLE_CHECK_FLAG</code>	Disables checking of the correctness of parameters of Data Fitting computational routines.

Use `DF_CHECK_FLAG` for `val_attr` in order to control validation of parameters of Data Fitting computational routines such as `df?Construct1D`, `df?Interpolate1D`/`df?InterpolateEx1D`, and `df?SearchCells1D`/`df?SearchCellsEx1D`, which can perform better with a small number of interpolation sites or integration limits (fewer than one dozen). The default mode, with checking of parameters enabled, should be used as you develop a Data Fitting-based application. After you complete development you can disable parameter checking in order to improve the performance of your application.

If you modify the parameter describing dimensions of the arrays that hold the vector-valued function or spline coefficients in contiguous memory, you should call the `df?EditPtr` task editor with the corresponding pointers to the vector-valued function or spline coefficients even when this pointer remains unchanged. Call the `df?EditIdxPtr` editor if those arrays are stored in non-contiguous memory locations.

You must call the `dfiEditVal` task editor to edit the structure of the partition `DF_XHINT` every time you modify a partition using `df?EditPtr`, even if the structure of the partition remains the same.

`df?EditIdxPtr`

Modifies a pointer to the memory representing a coordinate of the data stored in matrix format.

Syntax

```
status = dfsEditIdxPtr(task, ptr_attr, idx, ptr)
```

```
status = dfdEditIdxPtr(task, ptr_attr, idx, ptr)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>DFTaskPtr</code>	Descriptor of the task.
<code>ptr_attr</code>	<code>const MKL_INT</code>	Type of the data to be modified. The parameter takes one of the values described in "Data Attributes Supported by the <code>df?EditIdxPtr</code> Task Editor" .
<code>idx</code>	<code>const MKL_INT</code>	Index of the coordinate whose pointer is to be modified.
<code>ptr</code>	<code>const float*</code> for <code>dfsEditIdxPtr</code>	Pointer to the data that holds values of coordinate <code>idx</code> . For details, see table "Data Attributes Supported by the <code>df?EditIdxPtr</code> Task Editor" .

Name	Type	Description
------	------	-------------

	const double* for dfdEditIdxPtr	
--	------------------------------------	--

Output Parameters

Name	Type	Description
------	------	-------------

status	int	
--------	-----	--

Status of the routine:

- DF_STATUS_OK if the routine execution completed successfully.
- Non-zero error code otherwise. See ["Task Status and Error Reporting"](#) for error code definitions.

Description

The routine modifies a pointer to the array that holds the *idx* coordinate of vector-valued function *y* or the pointer to the array of spline coefficients corresponding to the given coordinate.

You can use the editor if you need to pass into a Data Fitting task or modify the pointer to coordinates of the vector-valued function or spline coefficients held at non-contiguous memory locations. Do not use the editor for coordinates at contiguous memory locations in row-major format.

Before calling this editor, make sure that you have created and initialized the task using a task creation function or a relevant editor such as the generic or specific df?EditPPSpline1D editor.

Data Attributes Supported by the df?EditIdxPtr Task Editor

Data Attribute	Description
DF_Y	Vector-valued function <i>y</i>
DF_PP_SCOEFF	Piecewise polynomial spline coefficients

When using df?EditIdxPtr, you might receive an error code in the following cases:

- You passed an unsupported parameter value into the editor.
- The value of the index exceeds the predefined value that equals the dimension *ny* of the vector-valued function.
- You pass a NULL pointer to the editor. In this case, the task remains unchanged.
- You pass a pointer to the *idx* coordinate of the vector-valued function you provided to contiguous memory in column-major format.

The code example below demonstrates how to use the editor for providing values of a vector-valued function stored in two non-contiguous arrays:

```
#define NX 1000 /* number of break points */
#define NY 2 /* dimension of vector-valued function */
int main()
{
    DFTaskPtr task;
    double x[NX];
    double y1[NX], y2[NX]; /* vector-valued function is stored as two arrays */
    /* Provide first coordinate of two-dimensional function y into creation routine */
    status = dfdNewTask1D( &task, NX, x, DF_NON_UNIFORM_PARTITION, NY, y1,
                          DF_1ST_COORDINATE );
    /* Provide second coordiante of two-dimensional function */
}
```



```

    status = dfdEditIdxPtr(task, DF_Y, 1, y2 );
    ...
}

```

df?QueryPtr

Reads a pointer to an array held in a Data Fitting task descriptor.

Syntax

```
status = dfsQueryPtr(task, ptr_attr, ptr)
```

```
status = dfdQueryPtr(task, ptr_attr, ptr)
```

Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	DFTaskPtr	Descriptor of the task.
<i>ptr_attr</i>	const MKL_INT	The parameter to query. The query routine supports pointer attributes described in the table " Pointers Supported by the df?EditPtr Task Editor ". For details, see the <i>Pointer Attribute</i> column in the table.

Output Parameters

Name	Type	Description
<i>ptr</i>	float** for dfsQueryPtr double** for dfdQueryPtr	Pointer to array returned by the query routine. For details, see the <i>Purpose</i> column in table " Pointers Supported by the df?EditPtr Task Editor ".
<i>status</i>	int	Status of the routine: <ul style="list-style-type: none"> • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The df?QueryPtr routine returns the pointer of type *ptr_attr* stored in a Data Fitting task descriptor as parameter *ptr*. Attributes of the pointers supported by the query function are identical to those supported by the editor df?EditPtr editor in the table "[Pointers Supported by the df?EditPtr Task Editor](#)".

You can use df?QueryPtr to read different types of pointers including pointers to the vector-valued function and spline coefficients stored in contiguous memory.

dfiQueryVal

Reads a parameter value in a Data Fitting task descriptor.

Syntax

```
status = dfiQueryVal(task, val_attr, val)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>DFTaskPtr</code>	Descriptor of the task.
<code>val_attr</code>	<code>const MKL_INT</code>	The parameter to query. The query function supports the parameter attributes described in "Parameters Supported by the dfiEditVal Task Editor" .

Output Parameters

Name	Type	Description
<code>val</code>	<code>MKL_INT</code>	The parameter value returned by the query function. See table "Parameters Supported by the dfiEditVal Task Editor" .
<code>status</code>	<code>int</code>	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The `dfiQueryVal` routine returns a parameter of type `val_attr` stored in a Data Fitting task descriptor as parameter `val`. The query function supports the parameter attributes described in ["Parameters Supported by the dfiEditVal Task Editor"](#).

`df?QueryIdxPtr`

Reads a pointer to the memory representing a coordinate of the data stored in matrix format.

Syntax

```
status = dfsQueryIdxPtr(task, ptr_attr, idx, ptr)
```

```
status = dfdQueryIdxPtr(task, ptr_attr, idx, ptr)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>DFTaskPtr</code>	Descriptor of the task.

Name	Type	Description
<code>ptr_attr</code>	<code>const MKL_INT</code>	Pointer attribute to query. The parameter takes one of the attributes described in "Data Attributes Supported by the df?EditIdxPtr Task Editor" .
<code>idx</code>	<code>const MKL_INT</code>	Index of the coordinate of the pointer to query.

Output Parameters

Name	Type	Description
<code>ptr</code>	<code>float*</code> for <code>dfsQueryIdxPtr</code> <code>double*</code> for <code>dfdQueryIdxPtr</code>	Pointer to the data that holds values of coordinate <code>idx</code> returned. For details, see table "Data Attributes Supported by the df?EditIdxPtr Task Editor" .
<code>status</code>	<code>int</code>	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The routine returns a pointer to the array that holds the `idx` coordinate of vector-valued function `y` or the pointer to the array of spline coefficients corresponding to the given coordinate.

You can use the query routine if you need the pointer to coordinates of the vector-valued function or spline coefficients held at non-contiguous memory locations or at a contiguous memory location in row-major format (the default storage format for spline coefficients).

Before calling this query routine, make sure that you have created and initialized the task using a task creation function or a relevant editor such as the generic or specific `df?EditPPSpline1D` editor.

When using `df?QueryIdxPtr`, you might receive an error code in the following cases:

- You passed an unsupported parameter value into the editor.
- The value of the index exceeds the predefined value that equals the dimension `ny` of the vector-valued function.
- You request the pointer to the `idx` coordinate of the vector-valued function you provided to contiguous memory in column-major format.

Data Fitting Computational Routines

Data Fitting computational routines are functions used to perform spline-based computations, such as:

- spline construction
- computation of values, derivatives, and integrals of the predefined order
- cell search

Once you create a Data Fitting task and initialize the required parameters, you can call computational routines as many times as necessary.

The table below lists the available computational routines:

Data Fitting Computational Routines

Routine	Description
df?Construct1D	Constructs a spline for a one-dimensional Data Fitting task.
df?Interpolate1D	Computes spline values and derivatives.
df?InterpolateEx1D	Computes spline values and derivatives by calling user-provided interpolants.
df?Integrate1D	Computes spline-based integrals.
df?IntegrateEx1D	Computes spline-based integrals by calling user-provided integrators.
df?SearchCells1D	Finds indices of cells containing interpolation sites.
df?SearchCellsEx1D	Finds indices of cells containing interpolation sites by calling user-provided cell searchers.

If a Data Fitting computation completes successfully, the computational routines return the `DF_STATUS_OK` code. If an error occurs, the routines return an error code specifying the origin of the failure. Some possible errors are the following:

- The task pointer is `NULL`.
- Memory allocation failed.
- The computation failed for another reason.

For the list of available status codes, see "[Task Status and Error Reporting](#)".

NOTE

Data Fitting computational routines do not control errors for floating-point conditions, such as overflow, gradual underflow, or operations with Not a Number (NaN) values.

[df?Construct1D](#)

Syntax

Constructs a spline of the given type.

```
status = dfsConstruct1D(task, s_format, method)
```

```
status = dfdConstruct1D(task, s_format, method)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>DFTaskPtr</code>	Descriptor of the task.
<code>s_format</code>	<code>const MKL_INT</code>	Spline format. The supported value is <code>DF_PP_SPLINE</code> .
<code>method</code>	<code>const MKL_INT</code>	Construction method. The supported value is <code>DF_METHOD_STD</code> .

Output Parameters

Name	Type	Description
<i>status</i>	int	Status of the routine: <ul style="list-style-type: none"> DF_STATUS_OK if the routine execution completed successfully. Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.

Description

Before calling `df?Construct1D`, you need to create and initialize the task, and set the parameters representing the spline. Then you can call the `df?Construct1D` routine to construct the spline. The format of the spline is defined by parameter *s_format*. The method for spline construction is defined by parameter *method*. Upon successful construction, the spline coefficients are available in the user-provided memory location in the format you set through the Data Fitting editor. For the available storage formats, see table ["Hint Values for Spline Coefficients"](#).

[df?Interpolate1D/df?InterpolateEx1D](#)

Runs data fitting computations.

Syntax

```
status = dfsInterpolate1D(task, type, method, nsite, site, sitehint, ndorder, dorder,
datahint, r, rhint, cell)
```

```
status = dfdInterpolate1D(task, type, method, nsite, site, sitehint, ndorder, dorder,
datahint, r, rhint, cell)
```

```
status = dfsInterpolateEx1D(task, type, method, nsite, site, sitehint, ndorder, dorder,
datahint, r, rhint, cell, le_cb, le_params, re_cb, re_params, i_cb, i_params, search_cb,
search_params)
```

```
status = dfdInterpolateEx1D(task, type, method, nsite, site, sitehint, ndorder, dorder,
datahint, r, rhint, cell, le_cb, le_params, re_cb, re_params, i_cb, i_params, search_cb,
search_params)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	DFTaskPtr	Descriptor of the task.
<i>type</i>	const MKL_INT	Type of spline-based computations. The parameter takes one or more values combined with an OR operation. For the list of possible values, see table "Computation Types Supported by the df?Interpolate1D/ df?Interpolate1D Routines" .
<i>method</i>	const MKL_INT	Computation method. The supported value is DF_METHOD_PP.

Name	Type	Description
<i>nsite</i>	const MKL_INT	Number of interpolation sites.
<i>site</i>	const float* for dfsInterpolate1D/ dfsInterpolateEx1D const double* for dfdInterpolate1D/ dfdInterpolateEx1D	<p>Array of interpolation sites of size <i>nsite</i>. The structure of the array is defined by the <i>sitehint</i> parameter:</p> <ul style="list-style-type: none"> • If sites form a non-uniform partition, the array should contain <i>nsite</i> values. • If sites form a uniform partition, the array should contain two entries that represent the left and the right interpolation sites. The first entry of the array contains the left-most interpolation point. The second entry of the array contains the right-most interpolation point.
<i>sitehint</i>	const MKL_INT	A flag describing the structure of the interpolation sites. For the list of possible values of <i>sitehint</i> , see table " Hint Values for Interpolation Sites ". If you set the flag to DF_NO_HINT, the library interprets the site-defined partition as non-uniform.
<i>ndorder</i>	const MKL_INT	Maximal derivative order increased by one to be computed at interpolation sites.
<i>dorder</i>	const MKL_INT*	Array of size <i>ndorder</i> that defines the order of the derivatives to be computed at the interpolation sites. If all the elements in <i>dorder</i> are zero, the library computes the spline values only. If you do not need interpolation computations, set <i>ndorder</i> to zero and pass a NULL pointer to <i>dorder</i> .
<i>datahint</i>	const float* for dfsInterpolate1D/ dfsInterpolateEx1D const double* for dfdInterpolate1D/ dfdInterpolateEx1D	Array that contains additional information about the structure of partition <i>x</i> and interpolation sites. This data helps to speed up the computation. If you provide a NULL pointer, the routine uses the default settings for computations. For details on the <i>datahint</i> array, see table " Structure of the datahint Array ".
<i>r</i>	float* for dfsInterpolate1D/ dfsInterpolateEx1D double* for dfdInterpolate1D/ dfdInterpolateEx1D	Array for results. If you do not need spline-based interpolation, set this pointer to NULL.
<i>rhint</i>	const MKL_INT	A flag describing the structure of the results. For the list of possible values of <i>rhint</i> , see table " Hint Values for the rhint Parameter ". If you set the flag to DF_NO_HINT, the library stores the result in row-major format.
<i>cell</i>	MKL_INT*	Array of cell indices in partition <i>x</i> that contain the interpolation sites. Provide this parameter as input if <i>type</i> is DF_INTERP_USER_CELL. If you do not need cell indices, set this parameter to NULL.

Name	Type	Description
<i>le_cb</i>	constdfsInterpCallBack for dfsInterpolateEx1D constdfdInterpCallBack for dfdInterpolateEx1D	User-defined callback function for extrapolation at the sites to the left of the interpolation interval. Set to NULL if you are not supplying a callback function.
<i>le_params</i>	const void*	Pointer to additional user-defined parameters passed by the library to the <i>le_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.
<i>re_cb</i>	constdfsInterpCallBack for dfsInterpolateEx1D constdfdInterpCallBack for dfdInterpolateEx1D	User-defined callback function for extrapolation at the sites to the right of the interpolation interval. Set to NULL if you are not supplying a callback function.
<i>re_params</i>	const void*	Pointer to additional user-defined parameters passed by the library to the <i>re_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.
<i>i_cb</i>	constdfsInterpCallBack for dfsInterpolateEx1D constdfdInterpCallBack for dfdInterpolateEx1D	User-defined callback function for interpolation within the interpolation interval. Set to NULL if you are not supplying a callback function.
<i>i_params</i>	const void*	Pointer to additional user-defined parameters passed by the library to the <i>i_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.
<i>search_cb</i>	constdfsSearchCellsCal lBack for dfsInterpolateEx1D constdfdSearchCellsCal lBack for dfdInterpolateEx1D	User-defined callback function for computing indices of cells that can contain interpolation sites. Set to NULL if you are not supplying a callback function.
<i>search_params</i>	const void*	Pointer to additional user-defined parameters passed by the library to the <i>search_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.

Output Parameters

Name	Type	Description
<i>status</i>	int	Status of the routine:

Name	Type	Description
		<ul style="list-style-type: none"> DF_STATUS_OK if the routine execution completed successfully. Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
<i>r</i>		Contains results of computations at the interpolation sites.
		Caution The <code>df?Interpolate1D/df?InterpolateEx1D</code> routines do not support in-place computations. You must provide non-aliasing memory locations for the <i>site</i> input parameter and the <i>r</i> output parameter.
<i>cell</i>		Array of cell indices in partition <i>x</i> that contain the interpolation sites, which is computed if <i>type</i> is DF_CELL.

Description

The `df?Interpolate1D/df?InterpolateEx1D` routine performs spline-based computations with user-defined settings. The routine supports two types of computations for interpolation sites provided in array *site*:

Computation Types Supported by the `df?Interpolate1D/df?InterpolateEx1D` Routines

Type	Description
DF_INTERP	Compute derivatives of predefined order. The derivative of the zero order is the spline value.
DF_INTERP_USER_CELL	Compute derivatives of predefined order given user-provided cell indices. The derivative of the zero order is the spline value. For this type of the computations you should provide a valid <i>cell</i> array, which holds the indices of cells in the <i>site</i> array containing relevant interpolation sites.
DF_CELL	Compute indices of cells in partition <i>x</i> that contain the sites.

If the indices of cells which contain interpolation types are available before the call to `df?Interpolate1D/df?InterpolateEx1D`, you can improve performance by using the DF_INTERP_USER_CELL computation type.

NOTE

If you pass any combination of `DF_INTERP`, `DF_INTERP_USER_CELL`, and `DF_CELL` computation types to the routine, the library uses the `DF_INTERP_USER_CELL` computation mode.

If you specify `DF_INTERP_USER_CELL` computation mode and a user-defined callback function for computing cell indices to `df?InterpolateEx1D`, the library uses the `DF_INTERP_USER_CELL` computation mode, and the call-back function is not called.

If the sites do not belong to interpolation interval $[a, b]$, the library uses:

- polynomial P_0 of the spline constructed on interval $[x_0, x_1]$ for computations at the sites to the left of a .
- polynomial P_{n-2} of the spline constructed on interval $[x_{n-2}, x_{n-1}]$ for computations at the sites to the right of b .

Interpolation sites support the following hints:

Hint Values for Interpolation Sites

Value	Description
<code>DF_NON_UNIFORM_PARTITION</code>	Partition is non-uniform.
<code>DF_UNIFORM_PARTITION</code>	Partition is uniform.
<code>DF_SORTED_DATA</code>	Interpolation sites are sorted in the ascending order and define a non-uniform partition.
<code>DF_NO_HINT</code>	No hint is provided. By default, the partition defined by interpolation sites is interpreted as non-uniform.

NOTE

If you pass a sorted array of interpolation sites to the Intel® oneAPI Math Kernel Library (oneMKL), set the `sitehint` parameter to the `DF_SORTED_DATA` value. The library uses this information when choosing the search algorithm and ignores any other data hints about the structure of the interpolation sites.

Data Fitting computation routines can use the following hints to speed up the computation:

- `DF_UNIFORM_PARTITION` describes the structure of breakpoints and the interpolation sites.
- `DF_QUASI_UNIFORM_PARTITION` describes the structure of breakpoints.

Pass the above hints to the library when appropriate.

For spline-based interpolation, you should set the derivatives whose values are required for the computation. You can provide the derivatives by setting the `dorder` array of size `ndorder` as follows:

$$dorder[i] = \begin{cases} 1, & \text{if derivative of the } i\text{-th order is required} \\ 0, & \text{otherwise} \end{cases} \quad i = 0, \dots, ndorder - 1$$

Orders of derivatives $i_d (d = 0, 1, \dots, nder - 1)$, corresponding to non-zero derivatives to be calculated, form the array $\{i_d\}$ of length $nder \leq ndorder$.

The storage format for the interpolation results is specified using the `rhint` parameter values. For each storage format, Table [Hint Values for the `rhint` Parameter](#) describes how to get the result $R(j, s, i_d)$ from array r , for function index $j (0 \leq j \leq yn - 1)$, site number $s (0 \leq s \leq nsite - 1)$, and derivative index $i_d (0 \leq i_d \leq nder - 1)$, where yn is the number of functions, $nsite$ is the number of sites, and $nder$ is the total number of non-zero derivatives for interpolation. The array r can be either a one-dimensional array of size $ny * nder * nsite$ or a three-dimensional array with the dimensions described in the table.

Hint Values for the *rhint* Parameter

Value	Location of $R(j, s, i_d)$, One-dimensional Array Storage	Location of $R(j, s, i_d)$, Three-dimensional Array Storage
DF_MATRIX_STORAGE_FU NCS_SITES_DERS (DF_MATRIX_STORAGE_ROWS)	$r[d + \text{nder} * (s + \text{nsite} * j)]$	$r[j][s][d]$ r declared as $r[\text{ny}][\text{nsite}][\text{nder}]$.
DF_MATRIX_STORAGE_FU NCS_DERS_SITES (DF_MATRIX_STORAGE_COLS)	$r[s + \text{nsite} * (d + \text{nder} * j)]$	$r[j][d][s]$ r declared as $r[\text{ny}][\text{nder}][\text{nsite}]$.
DF_MATRIX_STORAGE_SI TES_FUNCS_DERS	$r[d + \text{nder} * (j + \text{ny} * s)]$	$r[s][j][d]$ r declared as $r[\text{nsite}][\text{ny}][\text{nder}]$.
DF_MATRIX_STORAGE_SI TES_DERS_FUNCS	$r[j + \text{ny} * (d + \text{nder} * s)]$	$r[s][d][j]$ r declared as $r[\text{nsite}][\text{nder}][\text{ny}]$.
DF_NO_HINT	No hint is provided. By default, the results are stored in $\text{rhint} = \text{DF_MATRIX_STORAGE_FUNCS_SITES_DERS}$.	

The following figures show the structure of the storage formats. Each shows sequential memory layout line by line, left to right.

- Storage in r for $\text{rhint} = \text{DF_MATRIX_STORAGE_FUNCS_SITES_DERS}$ ($\text{DF_MATRIX_STORAGE_ROWS}$):

$R(0, 0, i_0)$	$R(0, 0, i_1)$...	$R(0, 0, i_{\text{nder} - 1})$
$R(0, 1, i_0)$	$R(0, 1, i_1)$...	$R(0, 1, i_{\text{nder} - 1})$
...
$R(0, \text{nsite} - 1, i_0)$	$R(0, \text{nsite} - 1, i_1)$...	$R(0, \text{nsite} - 1, i_{\text{nder} - 1})$
$R(1, 0, i_0)$	$R(1, 0, i_1)$...	$R(1, 0, i_{\text{nder} - 1})$
$R(1, 1, i_0)$	$R(1, 1, i_1)$...	$R(1, 1, i_{\text{nder} - 1})$
...
$R(1, \text{nsite} - 1, i_0)$	$R(1, \text{nsite} - 1, i_1)$...	$R(1, \text{nsite} - 1, i_{\text{nder} - 1})$
...

- Storage in r for $\text{rhint} = \text{DF_MATRIX_STORAGE_FUNCS_DERS_SITES}$ ($\text{DF_MATRIX_STORAGE_COLS}$):

$R(0, 0, i_0)$	$R(0, 1, i_0)$...	$R(0, nsite - 1, i_0)$
$R(0, 0, i_1)$	$R(0, 1, i_1)$...	$R(0, nsite - 1, i_1)$
...
$R(0, 0, i_{nder} - 1)$	$R(0, 1, i_{nder} - 1)$...	$R(0, nsite - 1, i_{nder} - 1)$

$R(1, 0, i_0)$	$R(1, 1, i_0)$...	$R(1, nsite - 1, i_0)$
$R(1, 0, i_1)$	$R(1, 1, i_1)$...	$R(1, nsite - 1, i_1)$
...
$R(1, 0, i_{nder} - 1)$	$R(1, 1, i_{nder} - 1)$...	$R(1, nsite - 1, i_{nder} - 1)$

...
-----	-----	-----	-----

- Storage in r for $rhint = \text{DF_MATRIX_STORAGE_SITES_FUNCS_DERS}$:

$R(0, 0, i_0)$	$R(0, 0, i_1)$...	$R(0, 0, i_{nder} - 1)$
$R(1, 0, i_0)$	$R(1, 0, i_1)$...	$R(1, 0, i_{nder} - 1)$
...
$R(ny - 1, 0, i_0)$	$R(ny - 1, 0, i_1)$...	$R(ny - 1, 0, i_{nder} - 1)$

$R(0, 1, i_0)$	$R(0, 1, i_1)$...	$R(0, 1, i_{nder} - 1)$
$R(1, 1, i_0)$	$R(1, 1, i_1)$...	$R(1, 1, i_{nder} - 1)$
...
$R(ny - 1, 1, i_0)$	$R(ny - 1, 1, i_1)$...	$R(ny - 1, 1, i_{nder} - 1)$

...
-----	-----	-----	-----

- Storage in r for $rhint = \text{DF_MATRIX_STORAGE_SITES_DERS_FUNCS}$:

$R(0, 0, i_0)$	$R(1, 0, i_0)$...	$R(ny - 1, 0, i_0)$
$R(0, 0, i_1)$	$R(1, 0, i_1)$...	$R(ny - 1, 0, i_1)$
...
$R(0, 0, i_{nder} - 1)$	$R(1, 0, i_{nder} - 1)$...	$R(ny - 1, 0, i_{nder} - 1)$

$R(0, 1, i_0)$	$R(1, 1, i_0)$...	$R(ny - 1, 1, i_0)$
$R(0, 1, i_1)$	$R(1, 1, i_1)$...	$R(ny - 1, 1, i_1)$
...
$R(0, 1, i_{nder} - 1)$	$R(1, 1, i_{nder} - 1)$...	$R(ny - 1, 1, i_{nder} - 1)$
...

To speed up Data Fitting computations, use the `datahint` parameter that provides additional information about the structure of the partition and interpolation sites. This data represents a floating-point or a double array with the following structure:

Structure of the `datahint` Array

Element Number	Description
0	Task dimension
1	Type of additional information
2	Reserved field
3	The total number q of elements containing additional information.
4	Element (1)
...	...
$q+3$	Element (q)

Data Fitting computation functions support the following types of additional information for `datahint[1]`:

Types of Additional Information

Type	Element Number	Parameter
DF_NO_APRIORI_INFO	0	No parameters are provided. Information about the data structure is absent.
DF_APRIORI_MOST_LIKELY_CELL	1	Index of the cell that is likely to contain interpolation sites.

To compute indices of the cells that contain interpolation sites, provide the pointer to the array of size `nsite` for the results. The library supports the following scheme of cell indexing for the given partition $\{x_i\}$, $i=1, \dots, nx$:

$cell[j] = i$, if $site[j] \in [x_i, x_{i+1})$, $i = 0, \dots, nx - 2$,

$cell[j] = nx - 1$, if $site[j] \in [x_{nx-1}, x_{nx}]$,

$cell[j] = nx$, if $site[j] \in (x_{nx}, x_{nx+1}]$,

where

- $x_0 = -\infty$
- $x_{nx+1} = +\infty$
- $j = 0, \dots, nsite-1$

To perform interpolation computations with spline types unsupported in the Data Fitting component, use the extended version of the routine `df?InterpolateEx1D`. With this routine, you can provide user-defined callback functions for computations within, to the left of, or to the right of interpolaton interval $[a, b]$. The callback functions compute indices of the cells that contain the specified interpolation sites or can serve as an approximation for computing the exact indices of such cells.

If you do not pass any function for computations at the sites outside the interval $[a, b]$, the routine uses the default settings.

See Also

Mathematical Conventions for Data Fitting Functions

[df?InterpCallback](#)

[df?SearchCellsCallback](#)

df?Integrate1D/df?IntegrateEx1D

Computes a spline-based integral.

Syntax

```
status = dfsIntegrate1D(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint,
rdatahint, r, rhint)
```

```
status = dfdIntegrate1D(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint,
rdatahint, r, rhint)
```

```
status = dfsIntegrateEx1D(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint,
rdatahint, r, rhint, le_cb, le_params, re_cb, re_params, i_cb, i_params, search_cb,
search_params)
```

```
status = dfdIntegrateEx1D(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint,
rdatahint, r, rhint, le_cb, le_params, re_cb, re_params, i_cb, i_params, search_cb,
search_params)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>DFTaskPtr</code>	Descriptor of the task.
<code>method</code>	<code>const MKL_INT</code>	Integration method. The supported value is <code>DF_METHOD_PP</code> .
<code>nlim</code>	<code>const MKL_INT</code>	Number of pairs of integration limits.
<code>llim</code>	<code>const float*</code> for <code>dfsIntegrate1D/</code> <code>dfsIntegrateEx1D</code> <code>const double*</code> for <code>dfdIntegrate1D/</code> <code>dfdIntegrateEx1D</code>	Array of size <code>nlim</code> that defines the left-side integration limits.

Name	Type	Description
<i>llimhint</i>	const MKL_INT	A flag describing the structure of the left-side integration limits <i>llim</i> . For the list of possible values of <i>llimhint</i> , see table "Hint Values for Integration Limits" . If you set the flag to the DF_NO_HINT value, the library assumes that the left-side integration limits define a non-uniform partition.
<i>rlim</i>	const float* for dfsIntegrate1D/ dfsIntegrateEx1D const double* for dfdIntegrate1D/ dfdIntegrateEx1D	Array of size <i>nlim</i> that defines the right-side integration limits.
<i>rlimhint</i>	const MKL_INT	A flag describing the structure of the right-side integration limits <i>rlim</i> . For the list of possible values of <i>rlimhint</i> , see table "Hint Values for Integration Limits" . If you set the flag to the DF_NO_HINT value, the library assumes that the right-side integration limits define a non-uniform partition.
<i>ldatahint</i>	const float* for dfsIntegrate1D/ dfsIntegrateEx1D const double* for dfdIntegrate1D/ dfdIntegrateEx1D	Array that contains additional information about the structure of partition <i>x</i> and left-side integration limits. For details on the <i>ldatahint</i> array, see table "Structure of the datahint Array" in the description of the df? Interpolate1D function.
<i>rdatahint</i>	const float* for dfsIntegrate1D/ dfsIntegrateEx1D const double* for dfdIntegrate1D/ dfdIntegrateEx1D	Array that contains additional information about the structure of partition <i>x</i> and right-side integration limits. For details on the <i>rdatahint</i> array, see table "Structure of the datahint Array" in the description of the df? Interpolate1D function.
<i>rhint</i>	const MKL_INT	A flag describing the structure of the results. For the list of possible values of <i>rhint</i> , see table "Hint Values for Integration Results" . If you set the flag to the DF_NO_HINT value, the library stores the results in row-major format.
<i>le_cb</i>	constdfsIntegrCallback for dfsIntegrateEx1D constdfdIntegrCallback for dfdIntegrateEx1D	User-defined callback function for integration on interval $[llim[i], \min(rlim[i], a))$ for $llim[i] < a$. Set to NULL if you are not supplying a callback function.
<i>le_params</i>	const void*	Pointer to additional user-defined parameters passed by the library to the <i>le_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.

Name	Type	Description
<i>re_cb</i>	<code>constdfsInterpCallBack</code> for <code>dfsIntegrateEx1D</code> <code>constdfdInterpCallBack</code> for <code>dfdIntegrateEx1D</code>	User-defined callback function for integration on interval $[\max(llim[i], b), rlim[i])$ for $rlim[i] \geq b$. Set to NULL if you are not supplying a callback function.
<i>re_params</i>	<code>const void*</code>	Pointer to additional user-defined parameters passed by the library to the <i>re_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.
<i>i_cb</i>	<code>constdfsIntegrCallBack</code> for <code>dfsIntegrateEx1D</code> <code>constdfdIntegrCallBack</code> for <code>dfdIntegrateEx1D</code>	User-defined callback function for integration on interval $[\max(a, llim[i],), \min(rlim[i], b))$. Set to NULL if you are not supplying a callback function.
<i>i_params</i>	<code>const void*</code>	Pointer to additional user-defined parameters passed by the library to the <i>i_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.
<i>search_cb</i>	<code>constdfsSearchCellsCal</code> <code>lBack</code> for <code>dfsIntegrateEx1D</code> <code>constdfdSearchCellsCal</code> <code>lBack</code> for <code>dfdIntegrateEx1D</code>	User-defined callback function for computing indices of cells that can contain interpolation sites. Set to NULL if you are not supplying a callback function.
<i>search_params</i>	<code>const void*</code>	Pointer to additional user-defined parameters passed by the library to the <i>search_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.

Output Parameters

Name	Type	Description
<i>status</i>	<code>int</code>	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
<i>r</i>	<code>float*</code> for <code>dfsIntegrate1D</code> / <code>dfsIntegrateEx1D</code> <code>double*</code> for <code>dfdIntegrate1D</code> / <code>dfdIntegrateEx1D</code>	Array of integration results. The size of the array should be sufficient to hold $nlim \cdot ny$ values, where <i>ny</i> is the dimension of the vector-valued function. The integration results are packed according to the settings in <i>rhint</i> .

Name	Type	Description
------	------	-------------

Caution

The `df?Integrate1D/df?IntegrateEx1D` routines do not support in-place computations. You must provide non-aliasing memory locations for the `llim` and `rlim` input parameters and the `r` output parameter.

Description

The `df?Integrate1D/df?IntegrateEx1D` routine computes spline-based integral on user-defined intervals

$$I(i, j) = \int_{ll_i}^{rl_i} f_j(x) dx$$

where $rl_i = rlim[i]$, $ll_i = llim[i]$, and $i = 0, \dots, ny - 1$.

If $rlim[i] < llim[i]$, the routine returns

$$I(i, j) = -\int_{rl_i}^{ll_i} f_j(x) dx$$

The routine supports the following hint values for integration results:

Hint Values for Integration Results

Value	Description
DF_MATRIX_STORAGE_ROWS	Data is stored in row-major format according to C conventions.
DF_MATRIX_STORAGE_COLS	Data is stored in column-major format according to Fortran conventions.
DF_NO_HINT	No hint is provided. By default, the coordinates of vector-valued function y are provided and stored in row-major format.

A common structure of the storage formats for the integration results is as follows:

- Row-major format

$I(0, 0)$...	$I(0, nlim - 1)$
...
$I(ny - 1, 0)$...	$I(ny - 1, nlim - 1)$

- Column-major format

$I(0, 0)$...	$I(ny - 1, 0)$
...
$I(0, nlim - 1)$...	$I(ny - 1, nlim - 1)$

Using the `llimhint` and `rlimhint` parameters, you can provide the following hint values for integration limits:

Hint Values for Integration Limits

Value	Description
<code>DF_SORTED_DATA</code>	Integration limits are sorted in the ascending order and define a non-uniform partition.
<code>DF_NON_UNIFORM_PARTITION</code>	Partition defined by integration limits is non-uniform.
<code>DF_UNIFORM_PARTITION</code>	Partition defined by integration limits is uniform.
<code>DF_NO_HINT</code>	No hint is provided. By default, partition defined by integration limits is interpreted as non-uniform.

To compute integration with splines unsupported in the Data Fitting component, use the extended version of the routine `df?IntegrateEx1D`. With this routine, you can provide user-defined callback functions that compute:

- integrals within, to the left of, or to the right of the interpolation interval $[a, b]$
- indices of cells that contain the provided integration limits or can serve as an approximation for computing the exact indices of such cells

If you do not pass callback functions, the routine uses the default settings.

See Also

Mathematical Conventions for Data Fitting Functions

[df?Interpolate1D/df?InterpolateEx1D](#)

[df?IntegrCallback](#)

[df?SearchCellsCallback](#)

[df?SearchCells1D/df?SearchCellsEx1D](#)

Searches sub-intervals containing interpolation sites.

Syntax

```
status = dfsSearchCells1D(task, method, nsite, site, sitehint, datahint, cell)
status = dfdSearchCells1D(task, method, nsite, site, sitehint, datahint, cell)
status = dfsSearchCellsEx1D(task, method, nsite, site, sitehint, datahint, cell,
search_cb, search_params)
status = dfdSearchCellsEx1D(task, method, nsite, site, sitehint, datahint, cell,
search_cb, search_params)
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>DFTaskPtr</code>	Descriptor of the task.

Name	Type	Description
<i>method</i>	const MKL_INT	Search method. The supported value is <code>DF_METHOD_STD</code> .
<i>nsite</i>	const MKL_INT*	Number of interpolation sites.
<i>site</i>	const float* for dfsSearchCells1D/ dfsSearchCellsEx1D const double* for dfdSearchCells1D/ dfdSearchCellsEx1D	<p>Array of interpolation sites of size <i>nsite</i>. The structure of the array is defined by the <i>sitehint</i> parameter:</p> <ul style="list-style-type: none"> • If the sites form a non-uniform partition, the array should contain <i>nsite</i> values. • If the sites form a uniform partition, the array should contain two entries that represent the left-most and the right-most interpolation sites. The first entry of the array contains the left-most interpolation point. The second entry of the array contains the right-most interpolation point.
<i>sitehint</i>	const MKL_INT	A flag describing the structure of the interpolation sites. For the list of possible values of <i>sitehint</i> , see table " Hint Values for Interpolation Sites ". If you set the flag to <code>DF_NO_HINT</code> , the library interprets the site-defined partition as non-uniform.
<i>datahint</i>	const float* for dfsSearchCells1D/ dfsSearchCellsEx1D const double* for dfdSearchCells1D/ dfdSearchCellsEx1D	Array that contains additional information about the structure of the partition and interpolation sites. This data helps to speed up the computation. If you provide a <code>NULL</code> pointer, the routine uses the default settings for computations. For details on the <i>datahint</i> array, see table " Structure of the datahint Array ".
<i>search_cb</i>	constdfsSearchCellsCallBac k for dfsSearchCellsEx1D constdfdSearchCellsCallBac k for dfdSearchCellsEx1D	<p>User-defined callback function for computing indices of cells that can contain interpolation sites.</p> <p>Set to <code>NULL</code> if you are not supplying a callback function.</p>
<i>search_params</i>	const void*	<p>Pointer to additional user-defined parameters passed by the library to the <i>search_cb</i> function.</p> <p>Set to <code>NULL</code> if there are no additional parameters or if you are not supplying a callback function.</p>

Output Parameters

Name	Type	Description
<i>status</i>	int	<p>Status of the routine:</p> <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
<i>cell</i>	MKL_INT*	Array of cell indices in the partition that contain the interpolation sites.

Description

The `df?SearchCells1D/df?SearchCellsEx1D` routines return array *cell* of indices of sub-intervals (cells) in the partition that contain interpolation sites available in array *site*. For details on the cell indexing scheme, see the description of the `df?Interpolate1D/df?InterpolateEx1D` computation routines.

Use the *datahint* parameter to provide additional information about the structure of the partition and/or interpolation sites. The definition of the *datahint* parameter is available in the description of the `df?Interpolate1D/df?InterpolateEx1D` computation routines.

For description of the user-defined callback for computation of cell indices, see `df?SearchCellsCallback`.

See Also

Mathematical Conventions for Data Fitting Functions

`df?Interpolate1D/df?InterpolateEx1D`

`df?SearchCellsCallback`

df?InterpCallback

A callback function for user-defined interpolation to be passed into `df?InterpolateEx1D`.

Syntax

```
status = dfsInterpCallback(n, cell, site, r, user_params, library_params)
```

```
status = dfdInterpCallback(n, cell, site, r, user_params, library_params)
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>long long*</code>	Number of interpolation sites.
<i>cell</i>	<code>long long*</code>	Array of size <i>n</i> containing indices of the cells to which the interpolation sites in array <i>site</i> belong.
<i>site</i>	<code>float*</code> for <code>dfsInterpCallback</code> <code>double*</code> for <code>dfdInterpCallback</code>	Array of interpolation sites of size <i>n</i> .
<i>user_params</i>	<code>void*</code>	Pointer to user-defined parameters of the callback function.
<i>library_params</i>	<code>dfInterpCallbackLibraryParams*</code>	Pointer to library-defined parameters of the callback function.

Output Parameters

Name	Type	Description
<i>status</i>	int	The status returned by the callback function: <ul style="list-style-type: none"> Zero indicates successful completion of the callback operation. A negative value indicates an error. A positive value indicates a warning. See "Task Status and Error Reporting" for error code definitions.
<i>r</i>	float* for dfsInterpCallback double* for dfdInterpCallback	Array of the computed interpolation results packed in row-major format.

Description

When passed into the `df?InterpolateEx1D` routine, this function performs user-defined interpolation operation.

The `library_params` parameter allows the library to provide extra parameters. Currently no parameters are provided.

See Also

[df?interpolate1d/df?interpolateex1d](#)

[df?searchcellscallback](#)

df?IntegrCallback

A callback function that you can pass into `df?IntegrateEx1D` to define integration computations.

Syntax

```
status = dfsIntegrCallback(n, lcell, llim, rcell, rlim, r, user_params, library_params)
```

```
status = dfdIntegrCallback(n, lcell, llim, rcell, rlim, r, user_params, library_params)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	long long*	Number of pairs of integration limits.
<i>lcell</i>	long long*	Array of size <i>n</i> with indices of the cells that contain the left-side integration limits in array <i>llim</i> .
<i>llim</i>	float* for dfsIntegrCallback double* for dfdIntegrCallback	Array of size <i>n</i> that holds the left-side integration limits.

Name	Type	Description
<i>rcell</i>	long long*	Array of size <i>n</i> with indices of the cells that contain the right-side integration limits in array <i>rlim</i> .
<i>rlim</i>	float* for <code>dfsIntegrCallback</code> double* for <code>dfdIntegrCallback</code>	Array of size <i>n</i> that holds the right-side integration limits.
<i>user_params</i>	void*	Pointer to user-defined parameters of the callback function.
<i>library_params</i>	<code>dfIntegrCallbackLibraryParams*</code>	Pointer to library-defined parameters of the callback function.

Output Parameters

Name	Type	Description
<i>status</i>	int	The status returned by the callback function: <ul style="list-style-type: none"> • Zero indicates successful completion of the callback operation. • A negative value indicates an error. • A positive value indicates a warning. See "Task Status and Error Reporting" for error code definitions.
<i>r</i>	float* for <code>dfsIntegrCallback</code> double* for <code>dfdIntegrCallback</code>	Array of integration results. For packing the results in row-major format, follow the instructions described in df?Interpolate1D/df?InterpolateEx1D .

Description

When passed into the `df?IntegrateEx1D` routine, this function defines integration computations. If at least one of the integration limits is outside the interpolation interval $[a, b]$, the library decomposes the integration into sub-intervals that belong to the extrapolation range to the left of *a*, the extrapolation range to the right of *b*, and the interpolation interval $[a, b]$, as follows:

- If the left integration limit is to the left of the interpolation interval ($llim < a$), the `df?IntegrateEx1D` routine passes *llim* as the left integration limit and $\min(rlim, a)$ as the right integration limit to the user-defined callback function.
- If the right integration limit is to the right of the interpolation interval ($rlim > b$), the `df?IntegrateEx1D` routine passes $\max(llim, b)$ as the left integration limit and *rlim* as the right integration limit to the user-defined callback function.
- If the left and the right integration limits belong to the interpolation interval, the `df?IntegrateEx1D` routine passes them to the user-defined callback function unchanged.

The value of the integral is the sum of integral values obtained on the sub-intervals.

The *library_params* parameter allows the library to provide extra parameters. Currently no parameters are provided.

Product and Performance Information

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

Notice revision #20201201

See Also

[df?Integrate1D/df?IntegrateEx1D](#)

[df?IntegrCallback](#)

[df?SearchCellsCallback](#)

df?SearchCellsCallback

A callback function for user-defined search to be passed into [df?InterpolateEx1D](#), [df?IntegrateEx1D](#), or [df?SearchCellsEx1D](#).

Syntax

```
status = dfsSearchCellsCallback(n, site, cell, flag, user_params, library_params)
```

```
status = dfdSearchCellsCallback(n, site, cell, flag, user_params, library_params)
```

Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	long long*	Number of interpolation sites or integration limits.
<i>site</i>	float* for dfsSearchCellsCallback double* for dfdSearchCellsCallback	Array, size <i>n</i> , of interpolation sites or integration limits.
<i>flag</i>	int*	Array of size <i>n</i> , with values set as follows: <ul style="list-style-type: none"> • If the cell with index <i>cell[i]</i> contains <i>site[i]</i>, set <i>flag[i]</i> to 1. • Otherwise, set <i>flag[i]</i> to zero. In this case, the library interprets the index as an approximation and computes the index of the cell containing <i>site[i]</i> by using the provided index as a starting point for the search.
<i>user_params</i>	void*	Pointer to user-defined parameters of the callback function.
<i>ms</i>		

Name	Type	Description
<code>library_params</code>	<code>dfSearchCallBackLibraryParams*</code>	Pointer to library-defined parameters of the callback function.

Output Parameters

Name	Type	Description
<code>status</code>	<code>int</code>	<p>The status returned by the callback function:</p> <ul style="list-style-type: none"> • Zero indicates successful completion of the callback operation. • A negative value indicates an error. • The <code>DF_STATUS_EXACT_RESULT</code> status indicates that cell indices returned by the callback function are exact. In this case, you do not need to initialize entries of the <code>flag</code> array. • A positive value indicates a warning. <p>See "Task Status and Error Reporting" for error code definitions.</p>
<code>cell</code>	<code>long long*</code>	Array of size <i>n</i> that returns indices of the cells computed by the callback function.

Description

When passed into the `df?InterpolateEx1D`, `df?IntegrateEx1D`, or `df?SearchCellsEx1D` routine, this function performs a user-defined search.

The `library_params` parameter allows the library to provide extra parameters. The `df?InterpolateEx1D`, and `df?SearchCellsEx1D` routines do not provide extra parameters and set `library_params` to `NULL`. The `df?IntegrateEx1D` routines use this parameter to specify which type of integration limits, left or right, are provided for the callback. To do this the library declares the `dfSearchCallBackLibraryParams` structure. It currently contains one field, `limit_type_flag`, of type `int`. The field is set by the library to one of two possible values: `DF_INTEGR_SEARCH_CB_LIM_FLAG` if the left integration limits are provided, or `DF_INTEGR_SEARCH_CB_RIM_FLAG` if the right integration limits are provided.

See Also

[df?Interpolate1D/df?InterpolateEx1D](#)

[df?InterpCallBack](#)

Data Fitting Task Destructors

Task destructors are routines used to delete task descriptors and deallocate the corresponding memory resources. The Data Fitting task destructor `dfDeleteTask` destroys a Data Fitting task and frees the memory.

`dfDeleteTask`

Destroys a Data Fitting task object and frees the memory.

Syntax

```
status = dfDeleteTask(&task)
```

Include Files

- `mk1.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>DFTaskPtr</code>	Descriptor of the task to destroy.

Output Parameters

Name	Type	Description
<code>status</code>	<code>int</code>	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the task is deleted successfully. • Non-zero error code if the operation failed. See "Task Status and Error Reporting" for error code definitions.

Description

Given a pointer to a task descriptor, this routine deletes the Data Fitting task descriptor and frees the memory allocated for the structure. If the task is deleted successfully, the routine sets the task pointer to `NULL`. Otherwise, the routine returns an error code.

Appendix A: Linear Solvers Basics

Many applications in science and engineering require the solution of a system of linear equations. This problem is usually expressed mathematically by the matrix-vector equation, $Ax = b$, where A is an m -by- n matrix, x is the n element column vector and b is the m element column vector. The matrix A is usually referred to as the coefficient matrix, and the vectors x and b are referred to as the solution vector and the right-hand side, respectively.

Basic concepts related to solving linear systems with sparse matrices are described in [Sparse Linear Systems](#) and various storage schemes for sparse matrices are described in [Sparse Matrix Storage Formats](#).

Sparse Linear Systems

In many real-life applications, most of the elements in A are zero. Such a matrix is referred to as sparse. Conversely, matrices with very few zero elements are called dense. For sparse matrices, computing the solution to the equation $Ax = b$ can be made much more efficient with respect to both storage and computation time, if the sparsity of the matrix can be exploited. The more an algorithm can exploit the sparsity without sacrificing the correctness, the better the algorithm.

Generally speaking, computer software that finds solutions to systems of linear equations is called a solver. A solver designed to work specifically on sparse systems of equations is called a sparse solver. Solvers are usually classified into two groups - direct and iterative.

Iterative Solvers start with an initial approximation to a solution and attempt to estimate the difference between the approximation and the true result. Based on the difference, an iterative solver calculates a new approximation that is closer to the true result than the initial approximation. This process is repeated until the difference between the approximation and the true result is sufficiently small. The main drawback to iterative solvers is that the rate of convergence depends greatly on the values in the matrix A . Consequently, it is not possible to predict how long it will take for an iterative solver to produce a solution. In fact, for ill-

conditioned matrices, the iterative process will not converge to a solution at all. However, for well-conditioned matrices it is possible for iterative solvers to converge to a solution very quickly. Consequently, if an application involves well-conditioned matrices iterative solvers can be very efficient.

Direct Solvers, on the other hand, factor the matrix A into the product of two triangular matrices and then perform a forward and backward triangular solve.

This approach makes the time required to solve a systems of linear equations relatively predictable, based on the size of the matrix. In fact, for sparse matrices, the solution time can be predicted based on the number of non-zero elements in the array A .

Matrix Fundamentals

A matrix is a rectangular array of either real or complex numbers. A matrix is denoted by a capital letter; its elements are denoted by the same lower case letter with row/column subscripts. Thus, the value of the element in row i and column j in matrix A is denoted by $a(i, j)$. For example, a 3 by 4 matrix A , is written as follows:

$$A = \begin{bmatrix} a(1, 1) & a(1, 2) & a(1, 3) & a(1, 4) \\ a(2, 1) & a(2, 2) & a(2, 3) & a(2, 4) \\ a(3, 1) & a(3, 2) & a(3, 3) & a(3, 4) \end{bmatrix}$$

Note that with the above notation, we assume the standard Fortran programming language convention of starting array indices at 1 rather than the C programming language convention of starting them at 0.

A matrix in which all of the elements are real numbers is called a real matrix. A matrix that contains at least one complex number is called a complex matrix. A real or complex matrix A with the property that $a(i, j) = a(j, i)$, is called a symmetric matrix. A complex matrix A with the property that $a(i, j) = \text{conj}(a(j, i))$, is called a Hermitian matrix. Note that programs that manipulate symmetric and Hermitian matrices need only store half of the matrix values, since the values of the non-stored elements can be quickly reconstructed from the stored values.

A matrix that has the same number of rows as it has columns is referred to as a square matrix. The elements in a square matrix that have same row index and column index are called the diagonal elements of the matrix, or simply the diagonal of the matrix.

The transpose of a matrix A is the matrix obtained by “flipping” the elements of the array about its diagonal. That is, we exchange the elements $a(i, j)$ and $a(j, i)$. For a complex matrix, if we both flip the elements about the diagonal and then take the complex conjugate of the element, the resulting matrix is called the Hermitian transpose or conjugate transpose of the original matrix. The transpose and Hermitian transpose of a matrix A are denoted by A^T and A^H respectively.

A column vector, or simply a vector, is a $n \times 1$ matrix, and a row vector is a $1 \times n$ matrix. A real or complex matrix A is said to be positive definite if the vector-matrix product $x^T A x$ is greater than zero for all non-zero vectors x . A matrix that is not positive definite is referred to as indefinite.

An upper (or lower) triangular matrix, is a square matrix in which all elements below (or above) the diagonal are zero. A unit triangular matrix is an upper or lower triangular matrix with all 1's along the diagonal.

A matrix P is called a permutation matrix if, for any matrix A , the result of the matrix product PA is identical to A except for interchanging the rows of A . For a square matrix, it can be shown that if PA is a permutation of the rows of A , then AP^T is the same permutation of the columns of A . Additionally, it can be shown that the inverse of P is P^T .

In order to save space, a permutation matrix is usually stored as a linear array, called a permutation vector, rather than as an array. Specifically, if the permutation matrix maps the i -th row of a matrix to the j -th row, then the i -th element of the permutation vector is j .

A matrix with non-zero elements only on the diagonal is called a diagonal matrix. As is the case with a permutation matrix, it is usually stored as a vector of values, rather than as a matrix.

Direct Method

For solvers that use the direct method, the basic technique employed in finding the solution of the system $Ax = b$ is to first factor A into triangular matrices. That is, find a lower triangular matrix L and an upper triangular matrix U , such that $A = LU$. Having obtained such a factorization (usually referred to as an LU decomposition or LU factorization), the solution to the original problem can be rewritten as follows.

$$\begin{aligned} Ax &= b \\ \Rightarrow LUx &= b \\ \Rightarrow L(Ux) &= b \end{aligned}$$

This leads to the following two-step process for finding the solution to the original system of equations:

1. Solve the systems of equations $Ly = b$.
2. Solve the system $Ux = y$.

Solving the systems $Ly = b$ and $Ux = y$ is referred to as a forward solve and a backward solve, respectively.

If a symmetric matrix A is also positive definite, it can be shown that A can be factored as LL^T where L is a lower triangular matrix. Similarly, a Hermitian matrix, A , that is positive definite can be factored as $A = LL^H$. For both symmetric and Hermitian matrices, a factorization of this form is called a Cholesky factorization.

In a Cholesky factorization, the matrix U in an LU decomposition is either L^T or L^H . Consequently, a solver can increase its efficiency by only storing L , and one-half of A , and not computing U . Therefore, users who can express their application as the solution of a system of positive definite equations will gain a significant performance improvement over using a general representation.

For matrices that are symmetric (or Hermitian) but not positive definite, there are still some significant efficiencies to be had. It can be shown that if A is symmetric but not positive definite, then A can be factored as $A = LDL^T$, where D is a diagonal matrix and L is a lower unit triangular matrix. Similarly, if A is Hermitian, it can be factored as $A = LDL^H$. In either case, we again only need to store L , D , and half of A and we need not compute U . However, the backward solve phases must be amended to solving $L^T x = D^{-1}y$ rather than $L^T x = y$.

Fill-In and Reordering of Sparse Matrices

Two important concepts associated with the solution of sparse systems of equations are fill-in and reordering. The following example illustrates these concepts.

Consider the system of linear equation $Ax = b$, where A is a symmetric positive definite sparse matrix, and A and b are defined by the following:

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ \frac{3}{2} & \frac{1}{2} & * & * & * \\ 6 & * & 12 & * & * \\ \frac{3}{4} & * & * & \frac{5}{8} & * \\ 3 & * & * & * & 16 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

A star (*) is used to represent zeros and to emphasize the sparsity of A . The Cholesky factorization of A is: $A = LL^T$, where L is the following:

$$L = \begin{bmatrix} 3 & * & * & * & * \\ \frac{1}{2} & \frac{1}{2} & * & * & * \\ 2 & -2 & 2 & * & * \\ \frac{1}{4} & \frac{1}{-4} & \frac{1}{-2} & \frac{1}{2} & * \\ 1 & -1 & -2 & -3 & 1 \end{bmatrix}$$

Notice that even though the matrix A is relatively sparse, the lower triangular matrix L has no zeros below the diagonal. If we computed L and then used it for the forward and backward solve phase, we would do as much computation as if A had been dense.

The situation of L having non-zeros in places where A has zeros is referred to as fill-in. Computationally, it would be more efficient if a solver could exploit the non-zero structure of A in such a way as to reduce the fill-in when computing L . By doing this, the solver would only need to compute the non-zero entries in L . Toward this end, consider permuting the rows and columns of A . As described in [Matrix Fundamentals](#), the permutations of the rows of A can be represented as a permutation matrix, P . The result of permuting the rows is the product of P and A . Suppose, in the above example, we swap the first and fifth row of A , then swap the first and fifth columns of A , and call the resulting matrix B . Mathematically, we can express the process of permuting the rows and columns of A to get B as $B = PAP^T$. After permuting the rows and columns of A , we see that B is given by the following:

$$B = \begin{bmatrix} 16 & * & * & * & 3 \\ * & \frac{1}{2} & * & * & \frac{3}{2} \\ * & * & 12 & * & 6 \\ * & * & * & \frac{5}{8} & \frac{3}{4} \\ 3 & \frac{3}{2} & 6 & \frac{3}{4} & 9 \end{bmatrix}$$

Since B is obtained from A by simply switching rows and columns, the numbers of non-zero entries in A and B are the same. However, when we find the Cholesky factorization, $B = LL^T$, we see the following:

$$L = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{3}}{4} \end{bmatrix}$$

The fill-in associated with B is much smaller than the fill-in associated with A . Consequently, the storage and computation time needed to factor B is much smaller than to factor A . Based on this, we see that an efficient sparse solver needs to find permutation P of the matrix A , which minimizes the fill-in for factoring $B = PAP^T$, and then use the factorization of B to solve the original system of equations.

Although the above example is based on a symmetric positive definite matrix and a Cholesky decomposition, the same approach works for a general LU decomposition. Specifically, let P be a permutation matrix, $B = PAP^T$ and suppose that B can be factored as $B = LU$. Then

$$Ax = b$$

$$\Rightarrow$$

$$PA(P^{-1}P)x = Pb$$

$$\Rightarrow$$

$$PA(P^TP)x = Pb$$

$$\Rightarrow$$

$$(PAP^T)(Px) = Pb$$

$$\Rightarrow$$

$$B(Px) = Pb$$

$$\Rightarrow$$

$$LU(Px) = Pb$$

It follows that if we obtain an LU factorization for B , we can solve the original system of equations by a three step process:

1. Solve $Ly = Pb$.
2. Solve $Uz = y$.
3. Set $x = P^T z$.

If we apply this three-step process to the current example, we first need to perform the forward solve of the systems of equation $Ly = Pb$:

$$Ly = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{3}}{4} \end{bmatrix} * \begin{bmatrix} y1 \\ y2 \\ y3 \\ y4 \\ y5 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 4 \\ 1 \end{bmatrix}$$

This gives:

$$y^T = \frac{5}{4}, 2\sqrt{2}, \frac{\sqrt{3}}{2}, \frac{16}{\sqrt{10}}, \frac{-979\sqrt{\frac{3}{5}}}{12}.$$

The second step is to perform the backward solve, $Uz = y$. Or, in this case, since a Cholesky factorization is used, $L^T z = y$.

$$\begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{5}}{4} \end{bmatrix}^T * \begin{bmatrix} z1 \\ z2 \\ z3 \\ z4 \\ z5 \end{bmatrix} = \begin{bmatrix} \frac{5}{4} \\ 2(\sqrt{2}) \\ \frac{\sqrt{3}}{2} \\ \frac{16}{\sqrt{10}} \\ \frac{-979\sqrt{3}}{12\sqrt{5}} \end{bmatrix}$$

This gives

$$z^T = \frac{123}{2}, 983, \frac{1961}{12}, 398, \frac{-979}{3}.$$

The third and final step is to set $x = P^T z$. This gives

$$X^T = \frac{-979}{3}, 983, \frac{1961}{12}, 398, \frac{123}{2}.$$

Sparse Matrix Storage Formats

It is more efficient to store only the non-zero elements of a sparse matrix. There are a number of common storage formats used for sparse matrices, but most of them employ the same basic technique. That is, store all non-zero elements of the matrix into a linear array and provide auxiliary arrays to describe the locations of the non-zero elements in the original matrix.

Storage Formats for the Direct Sparse Solvers

Storing the non-zero elements of a sparse matrix into a linear array is done by walking down each column (column-major format) or across each row (row-major format) in order, and writing the non-zero elements to a linear array in the order they appear in the walk.

- [DSS Symmetric Matrix Storage](#)
- [DSS Nonsymmetric Matrix Storage](#)
- [DSS Structurally Symmetric Matrix Storage](#)
- [DSS Distributed Symmetric Matrix Storage](#)

Sparse Matrix Storage Formats for Sparse BLAS Levels 2 and Level 3

These sections describe in detail the sparse matrix storage formats supported in the current version of the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 and Level 3.

- Sparse BLAS CSR Matrix Storage
- Sparse BLAS CSC Matrix Storage
- Sparse BLAS Coordinate Matrix Storage
- Sparse BLAS Diagonal Matrix Storage
- Sparse BLAS Skyline Matrix Storage
- Sparse BLAS BSR Matrix Storage

DSS Symmetric Matrix Storage

For symmetric matrices, it is necessary to store only the upper triangular half of the matrix (upper triangular format) or the lower triangular half of the matrix (lower triangular format).

The Intel® oneAPI Math Kernel Library (oneMKL) direct sparse solvers use a row-major upper triangular storage format: the matrix is compressed row-by-row and for symmetric matrices only non-zero elements in the upper triangular half of the matrix are stored.

The Intel® oneAPI Math Kernel Library (oneMKL) sparse matrix storage format for direct sparse solvers is specified by three arrays: *values*, *columns*, and *rowIndex*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix.

<i>values</i>	A real or complex array that contains the non-zero elements of a sparse matrix. The non-zero elements are mapped into the <i>values</i> array using the row-major upper triangular storage mapping described above.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column that contains the <i>i</i> -th element in the <i>values</i> array.
<i>rowIndex</i>	Element <i>j</i> of the integer array <i>rowIndex</i> gives the index of the element in the <i>values</i> array that is first non-zero element in a row <i>j</i> .

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in the matrix.

As the *rowIndex* array gives the location of the first non-zero element within a row, and the non-zero elements are stored consecutively, the number of non-zero elements in the *i*-th row is equal to the difference of *rowIndex*[*i*] and *rowIndex*[*i*+1].

To have this relationship hold for the last row of the matrix, an additional entry (dummy entry) is added to the end of *rowIndex*. Its value is equal to the number of non-zero elements plus one. This makes the total length of the *rowIndex* array one larger than the number of rows in the matrix.

NOTE

The Intel® oneAPI Math Kernel Library (oneMKL) sparse storage scheme for the direct sparse solvers supports both one-based indexing and zero-based indexing.

Consider the symmetric matrix *A*:

$$A = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -3 & * & 6 & 7 & * \\ * & * & 4 & * & -5 \end{pmatrix}$$

Only elements from the upper triangle are stored. The actual arrays for the matrix *A* are as follows:

Storage Arrays for a Symmetric Matrix

one-based indexing

<i>values</i>	=	(1	-1	-3	5	4	6	4	7	-5)
<i>columns</i>	=	(1	2	4	2	3	4	5	4	5)
<i>rowIndex</i>	=	(1	4	5	8	9	10)			

zero-based indexing

<i>values</i>	=	(1	-1	-3	5	4	6	4	7	-5)
<i>columns</i>	=	(0	1	3	1	2	3	4	3	4)
<i>rowIndex</i>	=	(0	3	4	7	8	9)			

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

DSS Nonsymmetric Matrix Storage

For a non-symmetric or non-Hermitian matrix, all non-zero elements need to be stored. Consider the non-symmetric matrix *B*:

$$\begin{pmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{pmatrix}$$

The matrix *B* has 13 non-zero elements, and all of them are stored as follows:

Storage Arrays for a Non-Symmetric Matrix

one-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
<i>rowIndex</i>	=	(1	4	6	9	12	14)							

zero-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
<i>rowIndex</i>	=	(0	3	5	8	11	13)							

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

DSS Structurally Symmetric Matrix Storage

Direct sparse solvers can also solve symmetrically structured systems of equations. A symmetrically structured system of equations is one where the pattern of non-zero elements is symmetric. That is, a matrix has a symmetric structure if $a_{j,i}$ is not zero if and only if $a_{i,j}$ is not zero. From the point of view of the solver software, a "non-zero" element of a matrix is any element stored in the *values* array, even if its value is

equal to 0. In that sense, any non-symmetric matrix can be turned into a symmetrically structured matrix by carefully adding zeros to the *values* array. For example, the above matrix *B* can be turned into a symmetrically structured matrix by adding two non-zero entries:

$$B = \begin{pmatrix} 1 & -1 & * & 3 & * \\ -2 & 5 & * & * & 0 \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & 0 & * & -5 \end{pmatrix}$$

The matrix *B* can be considered to be symmetrically structured with 15 non-zero elements and represented as:

Storage Arrays for a Symmetrically Structured Matrix

one-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)	
<i>columns</i>	=	(1	2	4	1	2	5	3	4	5	1	3	4	2	3	5)	
<i>rowIndex</i>	=	(1	4	7	10	13	16)										

zero-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)	
<i>columns</i>	=	(0	1	3	0	1	4	2	3	4	0	2	3	1	2	4)	
<i>rowIndex</i>	=	(0	3	6	9	12	15)										

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

DSS Distributed Symmetric Matrix Storage

The distributed assembled matrix input format can be used by the Parallel Direct Sparse Solver for Clusters Interface.

In this format, the symmetric input matrix *A* is divided into sequential row subsets, or domains. Each domain belongs to an MPI process. Neighboring domains can overlap. For such intersection between two domains, the element values of the full matrix can be obtained by summing the respective elements of both domains.

As in the centralized format, the distributed format uses three arrays to describe the input data, but the *values*, *columns*, and *rowIndex* arrays on each processor only describe the domain belonging to that particular processor and not the entire matrix.

For example, consider a symmetric matrix *A*:

$$A = \begin{pmatrix} 6 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 11 & 5 & 4 \\ -3 & * & 5 & 10 & * \\ * & * & 4 & * & 5 \end{pmatrix}$$

This array could be distributed between two domains corresponding to two MPI processes, with the first containing rows 1 through 3, and the second containing rows 3 through 5.

NOTE

For the symmetric input matrix, it is not necessary to store the values from the lower triangle.

$$A_{Domain1} = \begin{pmatrix} 6 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 3 & * & 2 \end{pmatrix}$$

Distributed Storage Arrays for a Symmetric Matrix, Domain 1**one-based indexing**

<i>values</i>	=	(6	-1	-3	5	3	2)
<i>columns</i>	=	(1	2	4	2	3	5)
<i>rowIndex</i>	=	(1	4	5	7)		

zero-based indexing

<i>values</i>	=	(6	-1	-3	5	3	2)
<i>columns</i>	=	(0	1	3	1	2	4)
<i>rowIndex</i>	=	(0	3	4	6)		

$$A_{Domain2} = \begin{pmatrix} * & * & 8 & 5 & 2 \\ -3 & * & 5 & 10 & * \\ * & * & 4 & * & 5 \end{pmatrix}$$

Distributed Storage Arrays for a Symmetric Matrix, Domain 2**one-based indexing**

<i>values</i>	=	(8	5	2	10	5)
<i>columns</i>	=	(3	4	5	4	5)
<i>rowIndex</i>	=	(1	4	5	6)	

zero-based indexing

<i>values</i>	=	(8	5	2	10	5)
<i>columns</i>	=	(2	3	4	3	4)
<i>rowIndex</i>	=	(0	3	4	5)	

The third row of matrix *A* is common between domain 1 and domain 2. The values of row 3 of matrix *A* are the sums of the respective elements of row 3 of matrix $A_{Domain1}$ and row 1 of matrix $A_{Domain2}$.

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

Product and Performance Information

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

Notice revision #20201201

Sparse BLAS CSR Matrix Storage Format

The Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS compressed sparse row (CSR) format is specified by four arrays:

- values*

- *columns*
- *pointerB*
- *pointerE*

In addition, each sparse matrix has an associated variable *indexing*, which specifies if the matrix indices are 0-based (*indexing*=0) or 1-based (*indexing*=1). These are descriptions of the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

<i>values</i>	A real or complex array that contains the non-zero elements of <i>A</i> . Values of the non-zero elements of <i>A</i> are mapped into the <i>values</i> array using the row-major storage mapping described above.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in <i>A</i> that contains the <i>i</i> -th value in the <i>values</i> array.
<i>pointerB</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>values</i> array that is first non-zero element in a row <i>j</i> of <i>A</i> . Note that this index is equal to <i>pointerB[j]-indexing</i> .
<i>pointerE</i>	An integer array that contains row indices, such that <i>pointerE[j]-1-indexing</i> is the index of the element in the <i>values</i> array that is last non-zero element in a row <i>j</i> of <i>A</i> .

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in *A*. The length of the *pointerB* and *pointerE* arrays is equal to the number of rows in *A*.

NOTE

Note that the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines support the CSR format both with one-based indexing and zero-based indexing.

You can represent the matrix *B*

$$B = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{pmatrix}$$

in the CSR format as:

Storage Arrays for a Matrix in CSR Format

one-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
<i>pointerB</i>	=	(1	4	6	9	12)								
<i>pointerE</i>	=	(4	6	9	12	14)								

zero-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
<i>pointerB</i>	=	(0	3	5	8	11)								
<i>pointerE</i>	=	(3	5	8	11	13)								

Additionally, you can define submatrices with different *pointerB* and *pointerE* arrays that share the same *values* and *columns* arrays of a CSR matrix. For example, you can represent the lower right 3x3 submatrix of *B* as:

Storage Arrays for a Matrix in CSR Format

one-based indexing

<i>subpointerB</i>	=	(6	10	13)
<i>subpointerE</i>	=	(9	12	14)

zero-based indexing

<i>subpointerB</i>	=	(5	9	12)
<i>subpointerE</i>	=	(8	11	13)

NOTE The CSR matrix must have a monotonically increasing row index. That is, $pointerB[i] \leq pointerB[j]$ and $pointerE[i] \leq pointerE[j]$ for all indices $i < j$.

This storage format is used in the NIST Sparse BLAS library [Rem05].

Three Array Variation of CSR Format

The storage format accepted for the direct sparse solvers is a variation of the CSR format. It also is used in the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS Level 2 both with one-based indexing and zero-based indexing. The above matrix *B* can be represented in this format (referred to as the 3-array variation of the CSR format or CSR3) as:

Storage Arrays for a Matrix in CSR Format (3-Array Variation)

one-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)	
<i>columns</i>	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)	
<i>rowIndex</i>	=	(1	4	6	9	12	14)								

zero-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)	
<i>columns</i>	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)	
<i>rowIndex</i>	=	(0	3	5	8	11	13)								

The 3-array variation of the CSR format has a restriction: all non-zero elements are stored continuously, that is the set of non-zero elements in the row *J* goes just after the set of non-zero elements in the row *J*-1.

There are no such restrictions in the general (NIST) CSR format. This may be useful, for example, if there is a need to operate with different submatrices of the matrix at the same time. In this case, it is enough to define the arrays *pointerB* and *pointerE* for each needed submatrix so that all these arrays are pointers to the same array *values*.

By definition, the array *rowIndex* from the Table "Storage Arrays for a Non-Symmetric Example Matrix" is related to the arrays *pointerB* and *pointerE* from the Table "Storage Arrays for an Example Matrix in CSR Format", and you can see that

```
pointerB[i] = rowIndex[i] for i=0, ..4;
pointerE[i] = rowIndex[i+1] for i=0, ..4.
```

This enables calling a routine that has *values*, *columns*, *pointerB* and *pointerE* as input parameters for a sparse matrix stored in the format accepted for the direct sparse solvers. For example, a routine with the interface:

```
void name_routine(..., double *values, MKL_INT *columns, MKL_INT *pointerB, MKL_INT
*pointerE, ...)
```

can be called with parameters *values*, *columns*, *rowIndex* as follows:

```
name_routine(..., values, columns, rowIndex, rowIndex+1, ...).
```

Sparse BLAS CSC Matrix Storage Format

The compressed sparse column format (CSC) is similar to the CSR format, but the columns are used instead the rows. In other words, the CSC format is identical to the CSR format for the transposed matrix. The CSR format is specified by four arrays: *values*, *columns*, *pointerB*, and *pointerE*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

<i>values</i>	A real or complex array that contains the non-zero elements of <i>A</i> . Values of the non-zero elements of <i>A</i> are mapped into the <i>values</i> array using the column-major storage mapping.
<i>rows</i>	Element <i>i</i> of the integer array <i>rows</i> is the number of the row in <i>A</i> that contains the <i>i</i> -th value in the <i>values</i> array.
<i>pointerB</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>values</i> array that is first non-zero element in a column <i>j</i> of <i>A</i> . Note that this index is equal to <i>pointerB[j]-indexing</i> for Inspector-executor Sparse BLAS CSC arrays.
<i>pointerE</i>	An integer array that contains column indices, such that <i>pointerE[j]-indexing</i> is the index of the element in the <i>values</i> array that is last non-zero element in a column <i>j</i> of <i>A</i> .

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in *A*. The length of the *pointerB* and *pointerE* arrays is equal to the number of columns in *A*.

NOTE

Note that the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines support the CSC format both with one-based indexing and zero-based indexing.

For example, consider matrix *B*:

$$B = \begin{pmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{pmatrix}$$

It can be represented in the CSC format as:

Storage Arrays for a Matrix in CSC Format

one-based indexing

<i>values</i>	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
<i>rows</i>	=	(1	2	4	1	2	5	3	4	1	3	4	3	5)
<i>pointerB</i>	=	(1	4	7	9	12)								
<i>pointerE</i>	=	(4	7	9	12	14)								

zero-based indexing

<i>values</i>	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
<i>rows</i>	=	(0	1	3	0	1	4	2	3	0	2	3	2	4)
<i>pointerB</i>	=	(0	3	6	8	11)								
<i>pointerE</i>	=	(3	6	8	11	13)								

Sparse BLAS Coordinate Matrix Storage Format

The coordinate format is the most flexible and simplest format for the sparse matrix representation. Only non-zero elements are stored, and the coordinates of each non-zero element are given explicitly. Many commercial libraries support the matrix-vector multiplication for the sparse matrices in the coordinate format.

The Intel® oneAPI Math Kernel Library (oneMKL) coordinate format is specified by three arrays: *values*, *rows*, and *column*, and a parameter *nnz* which is number of non-zero elements in *A*. All three arrays have dimension *nnz*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

<i>values</i>	A real or complex array that contains the non-zero elements of <i>A</i> in any order.
---------------	---

<i>rows</i>	Element <i>i</i> of the integer array <i>rows</i> is the number of the row in <i>A</i> that contains the <i>i</i> -th value in the <i>values</i> array.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in <i>A</i> that contains the <i>i</i> -th value in the <i>values</i> array.

NOTE

Note that the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines support the coordinate format both with one-based indexing and zero-based indexing.

For example, the sparse matrix *C*

$$C = \begin{pmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{pmatrix}$$

can be represented in the coordinate format as follows:

Storage Arrays for an Example Matrix in case of the coordinate format

one-based indexing														
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
rows	=	(1	1	1	2	2	3	3	3	4	4	4	5	5)
columns	=	(1	2	3	1	2	3	4	5	1	3	4	2	5)
zero-based indexing														
values	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
rows	=	(0	0	0	1	1	2	2	2	3	3	3	4	4)
columns	=	(0	1	2	0	1	2	3	4	0	2	3	1	4)

Sparse BLAS Diagonal Matrix Storage Format

If the sparse matrix has diagonals containing only zero elements, then the diagonal storage format can be used to reduce the amount of information needed to locate the non-zero elements. This storage format is particularly useful in many applications where the matrix arises from a finite element or finite difference discretization. The Intel® oneAPI Math Kernel Library (oneMKL) diagonal storage format is specified by two arrays: *values* and *distance*, and two parameters: *ndiag*, which is the number of non-empty diagonals, and *lval*, which is the declared leading dimension in the calling (sub)programs. The following table describes the arrays *values* and *distance*:

<i>values</i>	A real or complex two-dimensional array is dimensioned as <i>lval</i> by <i>ndiag</i> . Each column of it contains the non-zero elements of certain diagonal of <i>A</i> . The key point of the storage is that each element in <i>values</i> retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom. Note that the value of <i>distance</i> [<i>i</i>] is the number of elements to be padded for diagonal <i>i</i> .
<i>distance</i>	An integer array with dimension <i>ndiag</i> . Element <i>i</i> of the array <i>distance</i> is the distance between <i>i</i> -diagonal and the main diagonal. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

The above matrix *C* can be represented in the diagonal storage format as follows:

$$distance = (-3 \ -1 \ 0 \ 1 \ 2)$$

$$values = \begin{pmatrix} * & * & 1 & -1 & -3 \\ * & -2 & 5 & 0 & 0 \\ * & 0 & 4 & 6 & 4 \\ -4 & 2 & 7 & 0 & * \\ 8 & 0 & -5 & * & * \end{pmatrix}$$

where the asterisks denote padded elements.

When storing symmetric, Hermitian, or skew-symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For the Intel® oneAPI Math Kernel Library (oneMKL) triangular solver routines elements of the array *distance* must be sorted in increasing order. In all other cases the diagonals and distances can be stored in arbitrary order.

Sparse BLAS Skyline Matrix Storage Format

The skyline storage format is important for the direct sparse solvers, and it is well suited for Cholesky or LU decomposition when no pivoting is required.

The skyline storage format accepted in Intel® oneAPI Math Kernel Library (oneMKL) can store only triangular matrix or triangular part of a matrix. This format is specified by two arrays: *values* and *pointers*. The following table describes these arrays:

<i>values</i>	A scalar array. For a lower triangular matrix it contains the set of elements from each row of the matrix starting from the first non-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets.
<i>pointers</i>	An integer array with dimension $(m+1)$, where m is the number of rows for lower triangle (columns for the upper triangle). $pointers[i] - pointers[0] + 1$ gives the index of element in <i>values</i> that is first non-zero element in row (column) i . The value of $pointers[m]$ is set to $nnz + pointers[0]$, where nnz is the number of elements in the array <i>values</i> .

For example, consider the matrix C :

$$C = \begin{pmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{pmatrix}$$

The low triangle of the matrix C given above can be stored as follows:

```
values = [ 1  -2  5  4  -4  0  2  7  8  0  0  -5 ]
pointers = [ 0  1  3  4  8  12 ]
```

and the upper triangle of this matrix C can be stored as follows:

```
values = [ 1  -1  5  -3  0  4  6  7  4  0  -5 ]
pointers = [ 0  1  3  6  8  11 ]
```

This storage format is supported by the NIST Sparse BLAS library [Rem05].

Note that the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines operating with the skyline storage format do not support general matrices.

Sparse BLAS BSR Matrix Storage Format

The Intel® oneAPI Math Kernel Library (oneMKL) block compressed sparse row (BSR) format for sparse matrices is specified by four arrays: *values*, *columns*, *pointerB*, and *pointerE*. The following table describes these arrays.

<i>values</i>	A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block-by-block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them are equal to zero. Within each non-zero block elements are stored in column-major order in the case of one-based indexing, and in row-major order in the case of the zero-based indexing.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in the block matrix that contains the <i>i</i> -th non-zero block.
<i>pointerB</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>columns</i> array that is first non-zero block in a row <i>j</i> of the block matrix.
<i>pointerE</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>columns</i> array that contains the last non-zero block in a row <i>j</i> of the block matrix plus 1.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks. The length of the *pointerB* and *pointerE* arrays is equal to the number of block rows in the block matrix.

NOTE

Note that the Intel® oneAPI Math Kernel Library (oneMKL) Sparse BLAS routines support BSR format both with one-based indexing and zero-based indexing.

For example, consider the sparse matrix *D*

$$D = \begin{pmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ * & * & 1 & 4 & * & * \\ * & * & 5 & 1 & * & * \\ * & * & 4 & 3 & 7 & 2 \\ * & * & 0 & 0 & 0 & 0 \end{pmatrix}$$

If the size of the block equals 2, then the sparse matrix *D* can be represented as a 3x3 block matrix *E* with the following structure:

$$E = \begin{pmatrix} L & M & * \\ * & N & * \\ * & P & Q \end{pmatrix}$$

where

$$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, \quad M = \begin{pmatrix} 6 & 7 \\ 8 & 2 \end{pmatrix}, \quad N = \begin{pmatrix} 1 & 4 \\ 5 & 1 \end{pmatrix}, \quad P = \begin{pmatrix} 4 & 3 \\ 0 & 0 \end{pmatrix}, \quad Q = \begin{pmatrix} 7 & 2 \\ 0 & 0 \end{pmatrix}$$

The matrix *D* can be represented in the BSR format as follows:

one-based indexing

```
values = (1 2 0 1 6 8 7 2 1 5 4 1 4 0 3 0 7 0 2 0)
columns = (1 2 2 2 3)
pointerB = (1 3 4)
pointerE = (3 4 6)
```

zero-based indexing

```

values = [1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0]
columns = [0 1 1 1 2]
pointerB = [0 2 3]
pointerE = [2 3 5]

```

This storage format is supported by the NIST Sparse BLAS library [Rem05].

Three Array Variation of BSR Format

Intel® oneAPI Math Kernel Library (oneMKL) supports the variation of the BSR format that is specified by three arrays: *values*, *columns*, and *rowIndex*. The following table describes these arrays.

<i>values</i>	A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block by block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them is equal to zero. Within each non-zero block the elements are stored in column major order in the case of the one-based indexing, and in row major order in the case of the zero-based indexing.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in the block matrix that contains the <i>i</i> -th non-zero block.
<i>rowIndex</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>columns</i> array that is first non-zero block in a row <i>j</i> of the block matrix.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks.

As the *rowIndex* array gives the location of the first non-zero block within a row, and the non-zero blocks are stored consecutively, the number of non-zero blocks in the *i*-th row is equal to the difference of *rowIndex*[*i*] and *rowIndex*[*i*+1].

To retain this relationship for the last row of the block matrix, an additional entry (dummy entry) is added to the end of *rowIndex* with value equal to the number of non-zero blocks plus one. This makes the total length of the *rowIndex* array one larger than the number of rows of the block matrix.

The above matrix *D* can be represented in this 3-array variation of the BSR format as follows:

one-based indexing

```

values = [1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0]
columns = [1 2 2 2 3]
rowIndex = [1 3 4 6]

```

zero-based indexing

```

values = [1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0]
columns = [0 1 1 1 2]
rowIndex = [0 2 3 5]

```

When storing symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For example, consider the symmetric sparse matrix *F*:

$$F = \begin{pmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ 6 & 8 & 1 & 4 & * & * \\ 7 & 2 & 5 & 2 & * & * \\ * & * & * & * & 7 & 2 \\ * & * & * & * & 0 & 0 \end{pmatrix}$$

If the size of the block equals 2, then the sparse matrix F can be represented as a 3x3 block matrix G with the following structure:

$$G = \begin{pmatrix} L & M & * \\ M' & N & * \\ * & * & Q \end{pmatrix}$$

where

$$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, M = \begin{pmatrix} 6 & 7 \\ 8 & 2 \end{pmatrix}, M' = \begin{pmatrix} 6 & 8 \\ 7 & 2 \end{pmatrix}, N = \begin{pmatrix} 1 & 4 \\ 5 & 2 \end{pmatrix}, \text{ and } Q = \begin{pmatrix} 7 & 2 \\ 0 & 0 \end{pmatrix}$$

The symmetric matrix F can be represented in this 3-array variation of the BSR format (storing only the upper triangular part) as follows:

one-based indexing

```
values = [1 2 0 1 6 8 7 2 1 5 4 2 7 0 2 0]
columns = [1 2 2 3]
rowIndex = [1 3 4 5]
```

zero-based indexing

```
values = [1 0 2 1 6 7 8 2 1 4 5 2 7 2 0 0]
columns = [0 1 1 2]
rowIndex = [0 2 3 4]
```

Variable BSR Format

A variation of BSR3 is variable block compressed sparse row format. For a trust level t , $0 \leq t \leq 100$, rows similar up to t percent are placed in one supernode.

Appendix B: Routine and Function Arguments

The major arguments in the BLAS routines are vector and matrix, whereas VM functions work on vector arguments only. The sections that follow discuss each of these arguments and provide examples.

Vector Arguments in BLAS

Vector arguments are passed in one-dimensional arrays. The array dimension (length) and vector increment are passed as integer variables. The length determines the number of elements in the vector. The increment (also called stride) determines the spacing between vector elements and the order of the elements in the array in which the vector is passed.

A vector of length n and increment $incx$ is passed in a one-dimensional array x whose values are defined as

```
x[0], x[|incx|], ..., x[(n-1)* |incx|]
```

If *incx* is positive, then the elements in array *x* are stored in increasing order. If *incx* is negative, the elements in array *x* are stored in decreasing order with the first element defined as $x[(n-1) * |incx|]$. If *incx* is zero, then all elements of the vector have the same value, $x[0]$. The size of the one-dimensional array that stores the vector must always be at least

```
idimx = 1 + (n-1)* |incx|
```

Example. One-dimensional Real Array

Let $x[0:6]$ be the one-dimensional real array

```
x = [1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0].
```

If *incx* = 2 and *n* = 3, then the vector argument with elements in order from first to last is [1.0, 5.0, 9.0].

If *incx* = -2 and *n* = 4, then the vector elements in order from first to last is [13.0, 9.0, 5.0, 1.0].

If *incx* = 0 and *n* = 4, then the vector elements in order from first to last is [1.0, 1.0, 1.0, 1.0].

One-dimensional substructures of a matrix, such as the rows, columns, and diagonals, can be passed as vector arguments with the starting address and increment specified.

Storage of the *m*-by-*n* matrix can be based on either column-major ordering where the increment between elements in the same column is 1, the increment between elements in the same row is *m*, and the increment between elements on the same diagonal is *m* + 1; or row-major ordering where the increment between elements in the same row is 1, the increment between elements in the same column is *n*, and the increment between elements on the same diagonal is *n* + 1.

Example. Two-dimensional Real Matrix

Let *a* be a real 5 x 4 matrix declared as .

To scale the third column of *a* by 2.0, use the BLAS routine `sscal` with the following calling sequence:

```
cblas_sscal (5, 2.0, a[2], 4)
```

To scale the second row, use the statement:

```
cblas_sscal (4, 2.0, a[4], 1)
```

To scale the main diagonal of *a* by 2.0, use the statement:

```
cblas_sscal (4, 2.0, a[0], 5)
```

NOTE

The default vector argument is assumed to be 1.

Vector Arguments in Vector Math

Vector arguments of classic VM mathematical functions are passed in one-dimensional arrays with unit vector increment. It means that a vector of length *n* is passed contiguously in an array *a* whose values are defined as

```
a[0], a[1], ..., a[n-1].
```

Strided VM mathematical functions allow using positive increments for all input and output vector arguments.

To accommodate for arrays with other increments, or more complicated indexing, VM contains auxiliary Pack/Unpack functions that gather the array elements into a contiguous vector and then scatter them after the computation is complete.

Generally, if the vector elements are stored in a one-dimensional array a as

$a[m0], a[m1], \dots, a[mn-1]$

and need to be regrouped into an array y as

$y[k0], y[k1], \dots, y[kn-1],$

VM Pack/Unpack functions can use one of the following indexing methods:

Positive Increment Indexing

$kj = incy * j, mj = inca * j, j = 0, \dots, n-1.$

Constraint: $incy > 0$ and $inca > 0$.

For example, setting $incy = 1$ specifies gathering array elements into a contiguous vector.

This method is similar to that used in BLAS, with the exception that negative and zero increments are not permitted.

Index Vector Indexing

.

$kj = iy[j], mj = ia[j], j = 0, \dots, n-1.$

where ia and iy are arrays of length n that contain index vectors for the input and output arrays a and y , respectively.

Mask Vector Indexing

Indices kj, mj are such that:

.

$my[kj] \neq 0, ma[mj] \neq 0, j = 0, \dots, n-1.$

where ma and my are arrays that contain mask vectors for the input and output arrays a and y , respectively.

Vector Mathematical Functions

Matrix Arguments

Matrix arguments of the Intel® oneAPI Math Kernel Library routines can be stored in arrays, using the following storage schemes:

- [conventional full storage](#)
- [packed storage](#) for Hermitian, symmetric, or triangular matrices
- [band storage](#) for band matrices
- [rectangular full packed storage](#) for symmetric, Hermitian, or triangular matrices as compact as the Packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels.

Full storage is the simplest scheme. . A matrix A is stored in a one-dimensional array a , with the matrix element a_{ij} stored in the array element $a[i - 1 + (j - 1) * lda]$, where lda is the leading dimension of array a .

If a matrix is triangular (upper or lower, as specified by the argument *uplo*), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set.

Routines that handle symmetric or Hermitian matrices allow for either the upper or lower triangle of the matrix to be stored in the corresponding elements of the array:

if $uplo = 'U'$, a_{ij} is stored as described for $i \leq j$, other elements of a need not be set.

if $uplo = 'L'$, a_{ij} is stored as described for $j \leq i$, other elements of a need not be set.

Packed storage allows you to store symmetric, Hermitian, or triangular matrices more compactly: the relevant triangle (again, as specified by the argument $uplo$) is packed by columns in a one-dimensional array ap :

if $uplo = 'U'$, a_{ij} is stored in $ap[i - 1 + j(j - 1)/2]$ for $i \leq j$

if $uplo = 'L'$, a_{ij} is stored in $ap[i - 1 + (2*n - j)*(j - 1)/2]$ for $j \leq i$.

In descriptions of LAPACK routines, arrays with packed matrices have names ending in p .

Band storage is as follows: an m -by- n band matrix with kl non-zero sub-diagonals and ku non-zero super-diagonals is stored compactly in an array ab with $(kl+ku+1)*n$ elements. Thus,

a_{ij} is stored in $ab(ku+1+i-j,j)$ for $\max(1, j-ku) \leq i \leq \min(n, j+kl)$.

Use the band storage scheme only when kl and ku are much less than the matrix size n . Although the routines work correctly for all values of kl and ku , using the band storage is inefficient if your matrices are not really banded.

The band storage scheme is illustrated by the following example, when

$$m = n = 6, kl = 2, ku = 1$$

Array elements marked * are not used by the routines:

Banded matrix A						Band storage of A					
a_{11}	a_{12}	0	0	0	0	*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}
a_{21}	a_{22}	a_{23}	0	0	0	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}
a_{31}	a_{32}	a_{33}	a_{34}	0	0	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*
0	a_{42}	a_{43}	a_{44}	a_{45}	0	a_{31}	a_{42}	a_{53}	a_{64}	*	*
0	0	a_{53}	a_{54}	a_{55}	a_{56}						
0	0	0	a_{64}	a_{65}	a_{66}						

When a general band matrix is supplied for *LU factorization*, space must be allowed to store kl additional super-diagonals generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with $kl + ku$ super-diagonals. Thus,

a_{ij} is stored in $ab(kl+ku+1+i-j, j)$ for $\max(1, j-ku) \leq i \leq \min(n, j+kl)$.

The band storage scheme for LU factorization is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:

Banded matrix A						Band storage of A					
a_{11}	a_{12}	0	0	0	0	*	*	*	+	+	+
a_{21}	a_{22}	a_{23}	0	0	0	*	*	+	+	+	+
a_{31}	a_{32}	a_{33}	a_{34}	0	0	*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}
0	a_{42}	a_{43}	a_{44}	a_{45}	0	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}
0	0	a_{53}	a_{54}	a_{55}	a_{56}	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*
0	0	0	a_{64}	a_{65}	a_{66}	a_{31}	a_{42}	a_{53}	a_{64}	*	*

Array elements marked * are not used by the routines; elements marked + need not be set on entry, but are required by the LU factorization routines to store the results. The input array will be overwritten on exit by the details of the LU factorization as follows:

*	*	*	u_{14}	u_{25}	u_{36}
*	*	u_{13}	u_{24}	u_{35}	u_{46}
*	u_{12}	u_{23}	u_{34}	u_{45}	u_{56}
u_{11}	u_{22}	u_{33}	u_{44}	u_{55}	u_{66}
m_{21}	m_{32}	m_{43}	m_{54}	m_{65}	*
m_{31}	m_{42}	m_{53}	m_{64}	*	*

where u_{ij} are the elements of the upper triangular matrix U, and m_{ij} are the multipliers used during factorization.

Triangular band matrices are stored in the same format, with either $kl = 0$ if upper triangular, or $ku = 0$ if lower triangular. For symmetric or Hermitian band matrices with k sub-diagonals or super-diagonals, you need to store only the upper or lower triangle, as specified by the argument *uplo*:

if *uplo* = 'U', a_{ij} is stored in $ab(k+1+i-j,j)$ for $\max(1,j-k) \leq i \leq j$

if *uplo* = 'L', a_{ij} is stored in $ab(1+i-j,j)$ for $j \leq i \leq \min(n,j+k)$.

In descriptions of LAPACK routines, arrays that hold matrices in band storage have names ending in *b*.

In Fortran, column-major ordering of storage is assumed. This means that elements of the same column occupy successive storage locations.

Three quantities are usually associated with a two-dimensional array argument: its leading dimension, which specifies the number of storage locations between elements in the same row, its number of rows, and its number of columns. For a matrix in full storage, the leading dimension of the array must be at least as large as the number of rows in the matrix.

A character transposition parameter is often passed to indicate whether the matrix argument is to be used in normal or transposed form or, for a complex matrix, if the conjugate transpose of the matrix is to be used.

The values of the transposition parameter for these three cases are the following:

'N' or 'n'	normal (no conjugation, no transposition)
'T' or 't'	transpose
'C' or 'c'	conjugate transpose.

Example. Two-Dimensional Complex Array

Suppose A (1:5, 1:4) is the complex two-dimensional array presented by matrix

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) & (1.3, 0.13) & (1.4, 0.14) \\ (2.1, 0.21) & (2.2, 0.22) & (2.3, 0.23) & (1.4, 0.24) \\ (3.1, 0.31) & (3.2, 0.32) & (3.3, 0.33) & (1.4, 0.34) \\ (4.1, 0.41) & (4.2, 0.42) & (4.3, 0.43) & (1.4, 0.44) \\ (5.1, 0.51) & (5.2, 0.52) & (5.3, 0.53) & (1.4, 0.54) \end{bmatrix}$$

Let *transa* be the transposition parameter, *m* be the number of rows, *n* be the number of columns, and *lda* be the leading dimension. Then if

transa = 'N', *m* = 4, *n* = 2, and *lda* = 5, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) \\ (2.1, 0.21) & (2.2, 0.22) \\ (3.1, 0.31) & (3.2, 0.32) \\ (4.1, 0.41) & (4.2, 0.42) \end{bmatrix}$$

If *transa* = 'T', *m* = 4, *n* = 2, and *lda* = 5, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (2.1, 0.21) & (3.1, 0.31) & (4.1, 0.41) \\ (1.2, 0.12) & (2.2, 0.22) & (3.2, 0.32) & (4.2, 0.42) \end{bmatrix}$$

If $transa = 'C'$, $m = 4$, $n = 2$, and $lda = 5$, the matrix argument would be

$$\begin{bmatrix} (1.1, -0.11) & (2.1, -0.21) & (3.1, -0.31) & (4.1, -0.41) \\ (1.2, -0.12) & (2.2, -0.22) & (3.2, -0.32) & (4.2, -0.42) \end{bmatrix}$$

Note that care should be taken when using a leading dimension value which is different from the number of rows specified in the declaration of the two-dimensional array. For example, suppose the array A above is declared as a complex 5-by-4 matrix.

Then if $transa = 'N'$, $m = 3$, $n = 4$, and $lda = 4$, the matrix argument will be

$$\begin{bmatrix} (1.1, 0.11) & (5.1, 0.51) & (4.2, 0.42) & (3.3, 0.33) \\ (2.1, 0.21) & (1.2, 0.12) & (5.2, 0.52) & (4.3, 0.43) \\ (3.1, 0.31) & (2.2, 0.22) & (1.3, 0.13) & (5.3, 0.53) \end{bmatrix}$$

Rectangular Full Packed storage allows you to store symmetric, Hermitian, or triangular matrices as compact as the Packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels. To store an n -by- n triangle (and suppose for simplicity that n is even), you partition the triangle into three parts: two $n/2$ -by- $n/2$ triangles and an $n/2$ -by- $n/2$ square, then pack this as an n -by- $n/2$ rectangle (or $n/2$ -by- n rectangle), by transposing (or transpose-conjugating) one of the triangles and packing it next to the other triangle. Since the two triangles are stored in full storage, you can use existing efficient routines on them.

There are eight cases of RFP storage representation: when n is even or odd, the packed matrix is transposed or not, the triangular matrix is lower or upper. See below for all the eight storage schemes illustrated:

n is odd, A is lower triangular

Full format	RFP (not transposed)	RFP (transposed)
a_{11} X X X X X X	a_{11} a_{55} a_{65} a_{75}	
a_{21} a_{22} X X X X X	a_{21} a_{22} a_{66} a_{76}	
a_{31} a_{32} a_{33} X X X X	a_{31} a_{32} a_{33} a_{77}	a_{11} a_{21} a_{31} a_{41} a_{51} a_{61} a_{71}
a_{41} a_{42} a_{43} a_{44} X X X	a_{41} a_{42} a_{43} a_{44}	a_{55} a_{22} a_{32} a_{42} a_{52} a_{62} a_{72}
a_{51} a_{52} a_{53} a_{54} a_{55} X X	a_{51} a_{52} a_{53} a_{54}	a_{65} a_{66} a_{33} a_{43} a_{53} a_{63} a_{73}
a_{61} a_{62} a_{63} a_{64} a_{65} a_{66} X	a_{61} a_{62} a_{63} a_{64}	a_{75} a_{76} a_{77} a_{44} a_{54} a_{64} a_{74}
a_{71} a_{72} a_{73} a_{74} a_{75} a_{76} a_{77}	a_{71} a_{72} a_{73} a_{74}	

n is even, A is lower triangular

Full format	RFP (not transposed)	RFP (transposed)
a_{11} X X X X X	a_{44} a_{54} a_{64}	
a_{21} a_{22} X X X X	a_{11} a_{55} a_{65}	
a_{31} a_{32} a_{33} X X X	a_{21} a_{22} a_{66}	a_{44} a_{11} a_{21} a_{31} a_{41} a_{51} a_{61}
a_{41} a_{42} a_{43} a_{44} X X	a_{31} a_{32} a_{33}	a_{54} a_{55} a_{22} a_{32} a_{42} a_{52} a_{62}
a_{51} a_{52} a_{53} a_{54} a_{55} X	a_{41} a_{42} a_{43}	a_{64} a_{65} a_{66} a_{33} a_{43} a_{53} a_{63}
a_{61} a_{62} a_{63} a_{64} a_{65} a_{66}	a_{51} a_{52} a_{53}	
	a_{61} a_{62} a_{63}	

n is odd, A is upper triangular

Full format							RFP (not transposed)				RFP (transposed)						
a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{14}	a_{15}	a_{16}	a_{17}							
X	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{24}	a_{25}	a_{26}	a_{27}							
X	X	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{34}	a_{35}	a_{36}	a_{37}	a_{14}	a_{24}	a_{34}	a_{44}	a_{11}	a_{12}	a_{13}
X	X	X	a_{44}	a_{45}	a_{46}	a_{47}	a_{44}	a_{45}	a_{46}	a_{47}	a_{15}	a_{25}	a_{35}	a_{45}	a_{55}	a_{22}	a_{23}
X	X	X	X	a_{55}	a_{56}	a_{57}	a_{11}	a_{55}	a_{56}	a_{57}	a_{16}	a_{26}	a_{36}	a_{46}	a_{56}	a_{66}	a_{33}
X	X	X	X	X	a_{66}	a_{67}	a_{12}	a_{22}	a_{66}	a_{67}	a_{17}	a_{27}	a_{37}	a_{47}	a_{57}	a_{67}	a_{77}
X	X	X	X	X	X	a_{77}	a_{13}	a_{23}	a_{33}	a_{77}							

n is even, A is upper triangular

Full format						RFP (not transposed)			RFP (transposed)						
a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{14}	a_{15}	a_{16}							
X	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{24}	a_{25}	a_{26}							
X	X	a_{33}	a_{34}	a_{35}	a_{36}	a_{34}	a_{35}	a_{36}	a_{14}	a_{24}	a_{34}	a_{44}	a_{11}	a_{12}	a_{13}
X	X	X	a_{44}	a_{45}	a_{46}	a_{44}	a_{45}	a_{46}	a_{15}	a_{25}	a_{35}	a_{45}	a_{55}	a_{22}	a_{23}
X	X	X	X	a_{55}	a_{56}	a_{11}	a_{55}	a_{56}	a_{16}	a_{26}	a_{36}	a_{46}	a_{56}	a_{66}	a_{33}
X	X	X	X	X	a_{66}	a_{12}	a_{22}	a_{66}							
						a_{13}	a_{23}	a_{33}							

Intel® oneAPI Math Kernel Library (oneMKL) provides a number of routines such as `?hfrk`, `?sfrk` performing BLAS operations working directly on RFP matrices, as well as some conversion routines, for instance, `?tpttf` goes from the standard packed format to RFP and `?trttf` goes from the full format to RFP.

Please refer to the Netlib site for more information.

Note that in the descriptions of LAPACK routines, arrays with RFP matrices have names ending in `fp`.

Appendix C: FFTW Interface to Intel® Math Kernel Library

Intel® oneAPI Math Kernel Library (oneMKL) offers FFTW2 and FFTW3 interfaces to Intel® oneAPI Math Kernel Library (oneMKL) Fast Fourier Transform and Trigonometric Transform functionality. The purpose of these interfaces is to enable applications using FFTW (www.fftw.org) to gain performance with Intel® oneAPI Math Kernel Library (oneMKL) without changing the program source code.

Both FFTW2 and FFTW3 interfaces are provided in open source as FFTW wrappers to Intel® oneAPI Math Kernel Library (oneMKL). For ease of use, FFTW3 interface is also integrated in Intel® oneAPI Math Kernel Library (oneMKL).

FFTW Notational Conventions

This appendix typically employs path notations for Windows* OS.

FFTW2 Interface to Intel® oneAPI Math Kernel Library

This section describes a collection of C and Fortran wrappers providing FFTW 2.x interface to Intel® oneAPI Math Kernel Library (oneMKL). The wrappers translate calls to FFTW 2.x functions into the calls of the Intel® oneAPI Math Kernel Library (oneMKL) Fast Fourier Transform interface (FFT interface).

Note that Intel® oneAPI Math Kernel Library (oneMKL) FFT interface operates on both single- and double-precision floating-point data types.

Because of differences between FFTW and Intel® oneAPI Math Kernel Library (oneMKL) FFT functionalities, there are restrictions on using wrappers instead of the FFTW functions. Some FFTW functions have empty wrappers. However, many typical FFTs can be computed using these wrappers.

Refer to [Fourier Transform Functions](#), for better understanding the effects from the use of the wrappers.

Wrappers Reference

The section provides a brief reference for the FFTW 2.x C interface. For details please refer to the original FFTW 2.x documentation available at www.fftw.org.

Each FFTW function has its own wrapper. Some of them, which are *not* expressly listed in this section, are empty and do nothing, but they are provided to avoid link errors and satisfy the function calls.

See Also

[Limitations of the FFTW2 Interface to Intel® oneAPI Math Kernel Library \(oneMKL\)](#)

One-dimensional Complex-to-complex FFTs

The following functions compute a one-dimensional complex-to-complex Fast Fourier transform.

```
fftw_plan fftw_create_plan(int n, fftw_direction dir, int flags);
fftw_plan fftw_create_plan_specific(int n, fftw_direction dir, int flags, fftw_complex
*in, int istride, fftw_complex *out, int ostride);

void fftw(fftw_plan plan, int howmany, fftw_complex *in, int istride, int idist,
fftw_complex *out, int ostride, int odist);
void fftw_one(fftw_plan plan, fftw_complex *in , fftw_complex *out);

void fftw_destroy_plan(fftw_plan plan);
```

Multi-dimensional Complex-to-complex FFTs

The following functions compute a multi-dimensional complex-to-complex Fast Fourier transform.

```
fftwnd_plan fftwnd_create_plan(int rank, const int *n, fftw_direction dir, int flags);
fftwnd_plan fftw2d_create_plan(int nx, int ny, fftw_direction dir, int flags);
fftwnd_plan fftw3d_create_plan(int nx, int ny, int nz, fftw_direction dir, int flags);
fftwnd_plan fftwnd_create_plan_specific(int rank, const int *n, fftw_direction dir, int
flags, fftw_complex *in, int istride, fftw_complex *out, int ostride);
fftwnd_plan fftw2d_create_plan_specific(int nx, int ny, fftw_direction dir, int flags,
fftw_complex *in, int istride, fftw_complex *out, int ostride);
fftwnd_plan fftw3d_create_plan_specific(int nx, int ny, int nz, fftw_direction dir, int
flags, fftw_complex *in, int istride, fftw_complex *out, int ostride);

void fftwnd(fftwnd_plan plan, int howmany, fftw_complex *in, int istride, int idist,
fftw_complex *out, int ostride, int odist);
void fftwnd_one(fftwnd_plan plan, fftw_complex *in, fftw_complex *out);

void fftwnd_destroy_plan(fftwnd_plan plan);
```

One-dimensional Real-to-half-complex/Half-complex-to-real FFTs

Half-complex representation of a conjugate-even symmetric vector of size N in a real array of the same size N consists of $N/2+1$ real parts of the elements of the vector followed by non-zero imaginary parts in the reverse order. Because the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface does not currently support this representation, all wrappers of this kind are empty and do nothing.

Nevertheless, you can perform one-dimensional real-to-complex and complex-to-real transforms using `rfftwnd` functions with `rank=1`.

See Also

Multi-dimensional Real-to-complex/complex-to-real FFTs

Multi-dimensional Real-to-complex/Complex-to-real FFTs

The following functions compute multi-dimensional real-to-complex and complex-to-real Fast Fourier transforms.

```

rfftwnd_plan rfftwnd_create_plan(int rank, const int *n, fftw_direction dir, int flags);

rfftwnd_plan rfftw2d_create_plan(int nx, int ny, fftw_direction dir, int flags);

rfftwnd_plan rfftw3d_create_plan(int nx, int ny, int nz, fftw_direction dir, int flags);

rfftwnd_plan rfftwnd_create_plan_specific(int rank, const int *n, fftw_direction dir, int flags,
fftw_real *in, int istride, fftw_real *out, int ostride);

rfftwnd_plan rfftw2d_create_plan_specific(int nx, int ny, fftw_direction dir, int flags,
fftw_real *in, int istride, fftw_real *out, int ostride);

rfftwnd_plan rfftw3d_create_plan_specific(int nx, int ny, int nz, fftw_direction dir, int flags,
fftw_real *in, int istride, fftw_real *out, int ostride);

void rfftwnd_real_to_complex(rfftwnd_plan plan, int howmany, fftw_real *in, int istride,
int idist, fftw_complex *out, int ostride, int odist);

void rfftwnd_complex_to_real(rfftwnd_plan plan, int howmany, fftw_complex *in, int istride,
int idist, fftw_real *out, int ostride, int odist);

void rfftwnd_one_real_to_complex(rfftwnd_plan plan, fftw_real *in, fftw_complex *out);

void rfftwnd_one_complex_to_real(rfftwnd_plan plan, fftw_complex *in, fftw_real *out);

void rfftwnd_destroy_plan(rfftwnd_plan plan);

```

Multi-threaded FFTW

This section discusses multi-threaded FFTW wrappers only. MPI FFTW wrappers, available only with Intel® oneAPI Math Kernel Library (oneMKL) for the Linux* and Windows* operating systems, are described in [a separate section](#).

Unlike the original FFTW interface, every computational function in the FFTW2 interface to Intel® oneAPI Math Kernel Library (oneMKL) provides multithreaded computation by default, with the maximum number of threads permitted in FFT functions (see "Techniques to Set the Number of Threads" in *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*). To limit the number of threads, call the threaded FFTW computational functions:

```

void fftw_threads(int nthreads, fftw_plan plan, int howmany, fftw_complex *in, int istride,
int idist, fftw_complex *out, int ostride, int odist);

void fftw_threads_one(int nthreads, rfftwnd_plan plan, fftw_complex *in, fftw_complex *out);

...

void rfftwnd_threads_real_to_complex( int nthreads, rfftwnd_plan plan, int howmany,
fftw_real *in, int istride, int idist, fftw_complex *out, int ostride, int odist);

```

Compared to its non-threaded counterpart, every threaded computational function has `threads_` as the second part of its name and additional first parameter `nthreads`. Set the `nthreads` parameter to the thread limit to ensure that the computation requires at most that number of threads.

Product and Performance Information

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

Notice revision #20201201

FFTW Support Functions

The FFTW wrappers provide memory allocation functions to be used with FFTW:

```
void* fftw_malloc(size_t n);
```

```
void fftw_free(void* x);
```

The `fftw_malloc` wrapper aligns the memory on a 16-byte boundary.

If `fftw_malloc` fails to allocate memory, it aborts the application. To override this behavior, set a global variable `fftw_malloc_hook` and optionally the complementary variable `fftw_free_hook`:

```
void (*fftw_malloc_hook) (size_t n);
```

```
void (*fftw_free_hook) (void *p);
```

The wrappers use the function `fftw_die` to abort the application in cases when a caller cannot be informed of an error otherwise (for example, in computational functions that return `void`). To override this behavior, set a global variable `fftw_die_hook`:

```
void (*fftw_die_hook) (const char *error_string);
```

```
void fftw_die(const char *s);
```

Limitations of the FFTW2 Interface to Intel® oneAPI Math Kernel Library (oneMKL)

The FFTW2 wrappers implement the functionality of only those FFTW functions that Intel® oneAPI Math Kernel Library (oneMKL) can reasonably support. Other functions are provided as no-operation functions, whose only purpose is to satisfy link-time symbol resolution. Specifically, no-operation functions include:

- Real-to-half-complex and respective backward transforms
- Print plan functions
- Functions for importing/exporting/forgetting wisdom
- Most of the FFTW functions not covered by the original FFTW2 documentation

Because the Intel® oneAPI Math Kernel Library (oneMKL) implementation of FFTW2 wrappers does not use plan and plan node structures declared in `fftw.h`, the behavior of an application that relies on the internals of the plan structures defined in that header file is undefined.

FFTW2 wrappers define plan as a set of attributes, such as strides, used to commit the Intel® oneAPI Math Kernel Library (oneMKL) FFT descriptor structure. If an FFTW2 computational function is called with attributes different from those recorded in the plan, the function attempts to adjust the attributes of the plan and recommit the descriptor. So, repeated calls of a computational function with the same plan but different strides, distances, and other parameters may be performance inefficient.

Plan creation functions disregard most planner flags passed through the `flags` parameter. These functions take into account only the following values of `flags`:

- `FFTW_IN_PLACE`

If this value of `flags` is supplied, the plan is marked so that computational functions using that plan ignore the parameters related to output (`out`, `ostride`, and `odist`). Unlike the original FFTW interface, the wrappers never use the `out` parameter as a scratch space for in-place transforms.

- `FFTW_THREADSAFE`

If this value of `flags` is supplied, the plan is marked read-only. An attempt to change attributes of a read-only plan aborts the application.

FFTW wrappers are generally not thread safe. Therefore, do not use the same plan in parallel user threads simultaneously.

Installing FFTW2 Interface Wrappers

Wrappers are delivered as source code, which you must compile to build the wrapper library. Then you can substitute the wrapper and Intel® oneAPI Math Kernel Library (oneMKL) libraries for the FFTW library. The source code for the wrappers, makefiles, and files with lists of wrappers are located in the `.\interfaces\fftw2xc` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

Creating the Wrapper Library

Two header files are used to compile the C wrapper library: `fftw2_mkl.h` and `fftw.h`. The `fftw2_mkl.h` file is located in the `.\interfaces\fftw2xc\wrappers` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

The file `fftw.h`, used to compile libraries and located in the `.\include\fftw` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory, slightly differs from the original FFTW (www.fftw.org) header file `fftw.h`.

The source code for the wrappers, makefiles, and files with lists of functions are located in the `.\interfaces\fftw2xc` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

A wrapper library contains wrappers for complex and real transforms in a serial and multi-threaded mode for double- or single-precision floating-point data types. A makefile parameter manages the data type.

Parameters of a makefile also specify the platform (required), compiler, and data precision. The makefile comment heading provides the exact description of these parameters.

To build the library, run the `make` command on Linux* OS and macOS* or the `nmake` command on Windows* OS with appropriate parameters.

For example, on Linux OS the command

```
make libintel64
```

builds a double-precision wrapper library for Intel® 64 architecture based applications using the Intel® oneAPI DPC++/C++ Compiler or the Intel® Fortran Compiler (Compilers and data precision are chosen by default.)

Each makefile creates the library in the directory with Intel® oneAPI Math Kernel Library (oneMKL) libraries corresponding to the platform used. For example, `./lib/ia32` (on Linux OS and macOS) or `.\lib\ia32` (on Windows* OS).

In the names of a wrapper library, the suffix corresponds to the compiler used and the letter preceding the underscore is "c" for the C programming language.

For example,

```
fftw2xc_intel.lib (on Windows OS); libfftw2xc_intel.a (on Linux OS and macOS);
```

```
fftw2xc_ms.lib (on Windows OS); libfftw2xc_gnu.a (on Linux OS and macOS).
```

Application Assembling

Use the necessary original FFTW (www.fftw.org) header files without any modifications. Use the created wrapper library and the Intel® oneAPI Math Kernel Library (oneMKL) library instead of the FFTW library.

Running FFTW2 Interface Wrapper Examples

Intel® oneAPI Math Kernel Library (oneMKL) provides examples to demonstrate how to use the MPI FFTW wrapper library. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw2xc` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory. To build examples, several additional files are needed: `fftw.h`, `fftw_threads.h`,

`rfftw.h`, and `rfftw_threads.h`. These files are distributed with permission from FFTW and are available in `.\include\fftw`. The original files can also be found in FFTW 2.1.5 at <http://www.fftw.org/download.html>.

An example makefile uses the `function` parameter in addition to the parameters of the corresponding wrapper library makefile (see [Creating a Wrapper Library](#)). The makefile comment heading provides the exact description of these parameters.

An example makefile normally invokes examples. However, if the appropriate wrapper library is not yet created, the makefile first builds the library the same way as the wrapper library makefile does and then proceeds to examples.

If the parameter `function=<example_name>` is defined, only the specified example runs. Otherwise, all examples from the appropriate subdirectory run. The subdirectory `._results` is created, and the results are stored there in the `<example_name>.res` files.

Product and Performance Information

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

Notice revision #20201201

MPI FFTW2 Wrappers

MPI FFTW wrappers for FFTW 2 are available only with Intel® oneAPI Math Kernel Library (oneMKL) for the Linux* and Windows* operating systems.

MPI FFTW Wrappers Reference

The section provides a reference for MPI FFTW C interface.

Complex MPI FFTW

Complex One-dimensional MPI FFTW Transforms

```
fftw_mpi_plan fftw_mpi_create_plan(MPI_Comm comm, int n, fftw_direction dir, int
flags);

void fftw_mpi(fftw_mpi_plan p, int n_fields, fftw_complex *local_data, fftw_complex
*work);

void fftw_mpi_local_sizes(fftw_mpi_plan p, int *local_n, int *local_start, int
*local_n_after_transform, int *local_start_after_transform, int *total_local_size);

void fftw_mpi_destroy_plan(fftw_mpi_plan plan);
```

Argument restrictions:

- Supported values of `flags` are `FFTW_ESTIMATE`, `FFTW_MEASURE`, `FFTW_SCRAMBLED_INPUT` and `FFTW_SCRAMBLED_OUTPUT`. The same algorithm corresponds to all these values of the flags parameter. If any other `flags` value is supplied, the wrapper library reports an error 'CDFT error in wrapper: unknown flags'.
- The only supported value of `n_fields` is 1.

Complex Multi-dimensional MPI FFTW Transforms


```
fftwnd_mpi_plan fftw2d_mpi_create_plan(MPI_Comm comm, int nx, int ny, fftw_direction
dir, int flags);

fftwnd_mpi_plan fftw3d_mpi_create_plan(MPI_Comm comm, int nx, int ny, int nz,
fftw_direction dir, int flags);

fftwnd_mpi_plan fftwnd_mpi_create_plan(MPI_Comm comm, int dim, int *n, fftw_direction
dir, int flags);

void fftwnd_mpi(fftwnd_mpi_plan p, int n_fields, fftw_complex *local_data, fftw_complex
*work, fftwnd_mpi_output_order output_order);

void fftwnd_mpi_local_sizes(fftwnd_mpi_plan p, int *local_nx, int *local_x_start, int
*local_ny_after_transpose, int *local_y_start_after_transpose, int *total_local_size);

void fftwnd_mpi_destroy_plan(fftwnd_mpi_plan plan);
```

Argument restrictions:

- Supported values of *flags* are FFTW_ESTIMATE and FFTW_MEASURE. If any other value of *flags* is supplied, the wrapper library reports an error '*CDFT error in wrapper: unknown flags*'.
- The only supported value of *n_fields* is 1.

Real MPI FFTW

Real-to-Complex MPI FFTW Transforms

```
rfftwnd_mpi_plan rfftw2d_mpi_create_plan(MPI_Comm comm, int nx, int ny, fftw_direction
dir, int flags);

rfftwnd_mpi_plan rfftw3d_mpi_create_plan(MPI_Comm comm, int nx, int ny, int nz,
fftw_direction dir, int flags);

rfftwnd_mpi_plan rffwnd_mpi_create_plan(MPI_Comm comm, int dim, int *n, fftw_direction
dir, int flags);

void rfftwnd_mpi(rfftwnd_mpi_plan p, int n_fields, fftw_real *local_data, fftw_real
*work, fftwnd_mpi_output_order output_order);

void rfftwnd_mpi_local_sizes(rfftwnd_mpi_plan p, int *local_nx, int *local_x_start, int
*local_ny_after_transpose, int *local_y_start_after_transpose, int *total_local_size);

void rfftwnd_mpi_destroy_plan(rfftwnd_mpi_plan plan);
```

Argument restrictions:

- Supported values of *flags* are FFTW_ESTIMATE and FFTW_MEASURE. If any other value of *flags* is supplied, the wrapper library reports an error '*CDFT error in wrapper: unknown flags*'.
- The only supported value of *n_fields* is 1.

NOTE

- Function *rfftwnd_mpi_create_plan* can be used for both one-dimensional and multi-dimensional transforms.
 - Both values of the *output_order* parameter are supported: FFTW_NORMAL_ORDER and FFTW_TRANSPOSED_ORDER.
-

Creating MPI FFTW2 Wrapper Library

The source code for the wrappers, makefiles, and files with lists of wrappers are located in the `.\interfaces\fftw2x_cdft` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

A wrapper library contains C wrappers for Complex One-dimensional MPI FFTW Transforms and Complex Multi-dimensional MPI FFTW Transforms. The library also contains empty C wrappers for Real Multi-dimensional MPI FFTW Transforms. For details, see [MPI FFTW Wrappers Reference](#).

Parameters of a makefile specify the platform (required), compiler, and data precision. Specifying the platform is required. The makefile comment heading provides the exact description of these parameters.

To build the library, run the `make` command on Linux* OS and macOS* or the `nmake` command on Windows* OS with appropriate parameters.

For example, on Linux OS the command

```
make libintel64
```

builds a double-precision wrapper library for Intel® 64 architecture based applications using Intel MPI and the Intel® oneAPI DPC++/C++ Compiler (compilers and data precision are chosen by default.).

A makefile creates the wrapper library in the directory with the Intel® oneAPI Math Kernel Library (oneMKL) libraries corresponding to the used platform. For example, `./lib/ia32` (on Linux OS) or `./lib\ia32` (on Windows* OS).

In the wrapper library names, the suffix corresponds to the used data precision. For example,

`fftw2x_cdft_SINGLE.lib` on Windows OS;

`libfftw2x_cdft_DOUBLE.a` on Linux OS.

Application Assembling with MPI FFTW Wrapper Library

Use the necessary original FFTW (www.fftw.org) header files without any modifications. Use the created MPI FFTW wrapper library and the Intel® oneAPI Math Kernel Library (oneMKL) library instead of the FFTW library.

Running MPI FFTW2 Wrapper Examples

There are some examples that demonstrate how to use the MPI FFTW wrapper library for FFTW2. The source C code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw2x_cdft` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory. To build examples, one additional file, `fftw_mpi.h`, is needed. This file is distributed with permission from FFTW and is available in `.\include\fftw`. The original file can also be found in FFTW 2.1.5 at <http://www.fftw.org/download.html>.

Parameters for the example makefiles are described in the makefile comment headings and are similar to the parameters of the wrapper library makefiles (see [Creating MPI FFTW Wrapper Library](#)).

The table below lists examples available in the `.\examples\fftw2x_cdft\source` subdirectory.

Examples of MPI FFTW Wrappers

Source file for the example	Description
<code>wrappers_c1d.c</code>	One-dimensional Complex MPI FFTW transform, using <code>plan = fftw_mpi_create_plan(...)</code>
<code>wrappers_c2d.c</code>	Two-dimensional Complex MPI FFTW transform, using <code>plan = fftw2d_mpi_create_plan(...)</code>
<code>wrappers_c3d.c</code>	Three-dimensional Complex MPI FFTW transform, using <code>plan = fftw3d_mpi_create_plan(...)</code>
<code>wrappers_c4d.c</code>	Four-dimensional Complex MPI FFTW transform, using <code>plan = fftwnd_mpi_create_plan(...)</code>

Source file for the example	Description
<code>wrappers_r1d.c</code>	One-dimensional Real MPI FFTW transform, using <code>plan = rfftw_mpi_create_plan(...)</code>
<code>wrappers_r2d.c</code>	Two-dimensional Real MPI FFTW transform, using <code>plan = rfftw2d_mpi_create_plan(...)</code>
<code>wrappers_r3d.c</code>	Three-dimensional Real MPI FFTW transform, using <code>plan = rfftw3d_mpi_create_plan(...)</code>
<code>wrappers_r4d.c</code>	Four-dimensional Real MPI FFTW transform, using <code>plan = rfftwnd_mpi_create_plan(...)</code>

FFTW3 Interface to Intel® oneAPI Math Kernel Library

This section describes a collection of FFTW3 wrappers to Intel® oneAPI Math Kernel Library (oneMKL). The wrappers translate calls of FFTW3 functions to the calls of the Intel® oneAPI Math Kernel Library (oneMKL) Fourier transform (FFT) or Trigonometric Transform (TT) functions. The purpose of FFTW3 wrappers is to enable developers whose programs currently use the FFTW3 library to gain performance with the Intel® oneAPI Math Kernel Library (oneMKL) Fourier transforms without changing the program source code.

The FFTW3 wrappers provide a limited functionality compared to the original FFTW 3.x library, because of differences between FFTW and Intel® oneAPI Math Kernel Library (oneMKL) FFT and TT functionality. This section describes limitations of the FFTW3 wrappers and hints for their usage. Nevertheless, many typical FFT tasks can be performed using the FFTW3 wrappers to Intel® oneAPI Math Kernel Library (oneMKL).

The FFTW3 wrappers are integrated in Intel® oneAPI Math Kernel Library (oneMKL). The only change required to use Intel® oneAPI Math Kernel Library (oneMKL) through the FFTW3 wrappers is to link your application using FFTW3 against Intel® oneAPI Math Kernel Library (oneMKL).

A reference implementation of the FFTW3 wrappers is also provided in open source. You can find it in the `interfaces` directory of the Intel® oneAPI Math Kernel Library (oneMKL) distribution. You can use the reference implementation to create your own wrapper library (see [Building Your Own Wrapper Library](#))

See also these resources:

Intel® oneAPI Math Kernel Library (oneMKL) Release Notes	for the version of the FFTW3 library supported by the wrappers.
www.fftw.org	for a description of the FFTW interface.
Fourier Transform Functions	for a description of the Intel® oneAPI Math Kernel Library (oneMKL) FFT interface.
Trigonometric Transform Routines	for a description of Intel® oneAPI Math Kernel Library (oneMKL) TT interface.

Using FFTW3 Wrappers

The FFTW3 wrappers are a set of functions and data structures depending on one another. The wrappers are not designed to provide the interface on a function-per-function basis. Some FFTW3 wrapper functions are empty and do nothing, but they are present to avoid link errors and satisfy function calls.

This document does not list the declarations of the functions that the FFTW3 wrappers provide (you can find the declarations in the `fftw3.h` header file). Instead, this section comments on particular limitations of the wrappers and provides usage hints: . These are some known limitations of FFTW3 wrappers and their usage in Intel® oneAPI Math Kernel Library (oneMKL).

- The FFTW3 wrappers do not support long double precision because Intel® oneAPI Math Kernel Library (oneMKL) FFT functions operate only on single- and double-precision floating-point data types (`float` and `double`, respectively). Therefore the functions with prefix `fftwl_`, supporting the long double data type, are not provided.

- The wrappers provide equivalent implementation for double- and single-precision functions (those with prefixes `fftw_` and `fftwf_`, respectively). So, all these comments equally apply to the double- and single-precision functions and will refer to functions with prefix `fftw_`, that is, double-precision functions, for brevity.
- The FFTW3 interface that the wrappers provide is defined in the `fftw3.h` header file. This file is borrowed from the FFTW3.x package and distributed within Intel® oneAPI Math Kernel Library (oneMKL) with permission. Additionally, the `fftw3_mkl.h` header file defines supporting structures and supplementary constants and macros.
- Actual functionality of the plan creation wrappers is implemented in `guru64` set of functions. Basic interface, advanced interface, and `guru` interface plan creation functions call the `guru64` interface functions. So, all types of the FFTW3 plan creation interface in the wrappers are functional.
- Plan creation functions may return a `NULL` plan, indicating that the functionality is not supported. So, please carefully check the result returned by plan creation functions in your application. In particular, the following problems return a `NULL` plan:
 - `c2r` and `r2c` problems with a split storage of complex data.
 - `r2r` problems with `kind` values `FFTW_R2HC`, `FFTW_HC2R`, and `FFTW_DHT`. The only supported `r2r` kinds are even/odd DFTs (sine/cosine transforms).
 - Multidimensional `r2r` transforms.
 - Transforms of multidimensional vectors. That is, the only supported values for parameter `howmany_rank` in `guru` and `guru64` plan creation functions are 0 and 1.
 - Multidimensional transforms with `rank > MKL_MAXRANK`.
- The `MKL_RODFT00` value of the `kind` parameter is introduced by the FFTW3 wrappers. For better performance, you are strongly encouraged to use this value rather than `FFTW_RODFT00`. To use this `kind` value, provide an extra first element equal to 0.0 for the input/output vectors. Consider the following example:

```
plan1 = fftw_plan_r2r_1d(n, in1, out1, FFTW_RODFT00, FFTW_ESTIMATE);
plan2 = fftw_plan_r2r_1d(n, in2, out2, MKL_RODFT00, FFTW_ESTIMATE);
```

Both plans perform the same transform, except that the `in2/out2` arrays have one extra zero element at location 0. For example, if `n=3`, `in1={x,y,z}` and `out1={u,v,w}`, then `in2={0,x,y,z}` and `out2={0,u,v,w}`.

- The `flags` parameter in plan creation functions is always ignored. The same algorithm is used regardless of the value of this parameter. In particular, `flags` values `FFTW_ESTIMATE`, `FFTW_MEASURE`, etc. have no effect.
- For multithreaded plans, use normal sequence of calls to the `fftw_init_threads()` and `fftw_plan_with_nthreads()` functions (refer to FFTW documentation).
- Memory allocation function `fftw_malloc` returns memory aligned at a 16-byte boundary. You must free the memory with `fftw_free`.
- The FFTW3 wrappers to Intel® oneAPI Math Kernel Library (oneMKL) use the 32-bit `int` type in both LP64 and ILP64 interfaces of Intel® oneAPI Math Kernel Library (oneMKL). Use `guru64` FFTW3 interfaces for 64-bit sizes.
- The wrappers typically indicate a problem by returning a `NULL` plan. In a few cases, the wrappers may report a descriptive message of the problem detected. By default the reporting is turned off. To turn it on, set variable `fftw3_mkl.verbose` to a non-zero value, for example:

```
#include "fftw3.h"
#include "fftw3_mkl.h"
fftw3_mkl.verbose = 0;
plan = fftw_plan_r2r(...);
```

- The following functions are empty:
 - For saving, loading, and printing plans
 - For saving and loading wisdom
 - For estimating arithmetic cost of the transforms.
- Do not use macro `FFTW_DLL` with the FFTW3 wrappers to Intel® oneAPI Math Kernel Library (oneMKL).

- Do not use negative stride values. Though FFTW3 wrappers support negative strides in the part of advanced and guru FFTW interface, the underlying implementation does not.
- Do not set a FFTW2 wrapper library before a FFTW3 wrapper library or Intel® oneAPI Math Kernel Library (oneMKL) in your link line application. All libraries define "fftw_destroy_plan" symbol and linkage in incorrect order results into expected errors.

Building Your Own FFTW3 Interface Wrapper Library

The FFTW3 wrappers to Intel® oneAPI Math Kernel Library (oneMKL) are delivered both integrated in Intel® oneAPI Math Kernel Library (oneMKL) and as source code, which can be compiled to build a standalone wrapper library with exactly the same functionality. Normally you do not need to build the wrappers yourself.

The source code for the wrappers, makefiles, and files with lists of functions are located in the `.\interfaces\fftw3xc` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

To build the wrappers,

1. Change the current directory to the wrapper directory
2. Run the `make` command on Linux* OS and macOS* or the `nmake` command on Windows* OS with a required target and optionally several parameters.

The target `libia32` or `libintel64` defines the platform architecture, and the other parameters specify the compiler, size of the default integer type, and placement of the resulting wrapper library. You can find a detailed and up-to-date description of the parameters in the makefile.

In the following example, the `make` command is used to build the FFTW3 C wrappers to Intel® oneAPI Math Kernel Library (oneMKL) for use from the GNU `gcc`* compiler on Linux OS based on Intel® 64 architecture:

```
cd interfaces/fftw3xc
make libintel64 compiler=gnu INSTALL_DIR=/my/path
```

This command builds the wrapper library and places the result, named `libfftw3xc_gnu.a`, into the `/my/path` directory. The name of the resulting library is composed of the name of the compiler used and may be changed by an optional parameter `INSTALL_LIBNAME`.

Building an Application With FFTW3 Interface Wrappers

Normally, the only change needed to build your application with FFTW3 wrappers replacing original FFTW library is to add Intel® oneAPI Math Kernel Library (oneMKL) at the link stage (see section "*Linking Your Application with Intel® oneAPI Math Kernel Library*" in the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*).

If you recompile your application, add subdirectory `include\fftw` to the search path for header files to avoid FFTW3 version conflicts.

Sometimes, you may have to modify your application according to the following recommendations:

- The application requires


```
#include "fftw3.h",
```

 which it probably already includes.
- The application does not require


```
#include "mkl_dfti.h" .
```
- The application does not require


```
#include "fftw3_mkl.h" .
```

It is required only in case you want to use the `MKL_RODFT00` constant.

- If the application does not check whether a `NULL` plan is returned by plan creation functions, this check must be added, because the FFTW3 to Intel® oneAPI Math Kernel Library (oneMKL) wrappers do not provide 100% of FFTW3 functionality.

Running FFTW3 Interface Wrapper Examples

There are some examples that demonstrate how to use the wrapper library. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw3xc` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

Parameters of the example makefiles are similar to the parameters of the wrapper library makefiles. Example makefiles normally build and invoke the examples. If the parameter `function=<example_name>` is defined, then only the specified example will run. Otherwise, all examples will be executed. Results of running the examples are saved in subdirectory `._results` in files with extension `.res`.

For detailed information about options for the example makefile, refer to the makefile.

MPI FFTW3 Wrappers

This section describes a collection of MPI FFTW wrappers to Intel® oneAPI Math Kernel Library (oneMKL).

MPI FFTW wrappers are available only with Intel® oneAPI Math Kernel Library (oneMKL) for the Linux* and Windows* operating systems.

These wrappers translate calls of MPI FFTW functions to the calls of the Intel® oneAPI Math Kernel Library (oneMKL) cluster Fourier transform (CFFT) functions. The purpose of the wrappers is to enable users of MPI FFTW functions improve performance of the applications without changing the program source code.

Although the MPI FFTW wrappers provide less functionality than the original FFTW3 because of differences between MPI FFTW and Intel® oneAPI Math Kernel Library (oneMKL) CFFT, the wrappers cover many typical CFFT use cases.

The MPI FFTW wrappers are provided as source code. To use the wrappers, you need to build your own wrapper library (see [Building Your Own Wrapper Library](#)).

See also these resources:

Intel® oneAPI Math Kernel Library (oneMKL) Release Notes	for the version of the FFTW3 library supported by the wrappers.
www.fftw.org	for a description of the MPI FFTW interface.
Cluster FFT Functions	for a description of the Intel® oneAPI Math Kernel Library (oneMKL) CFFT interface.

Building Your Own Wrapper Library

The MPI FFTW wrappers for FFTW3 are delivered as source code, which can be compiled to build a wrapper library.

The source code for the wrappers, makefiles, and files with lists of functions are located in subdirectory `.\interfaces\fftw3x_cdft` in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

To build the wrappers,

1. Change the current directory to the wrapper directory
2. Run the `make` command on Linux* OS or the `nmake` command on Windows* OS with a required target and optionally several parameters.

The target `libia32` or `libintel64` defines the platform architecture, and the other parameters specify the compiler, size of the default `INTEGER` type, as well as the name and placement of the resulting wrapper library. You can find a detailed and up-to-date description of the parameters in the makefile.

In the following example, the `make` command is used to build the MPI FFTW wrappers to Intel® oneAPI Math Kernel Library (oneMKL) for use from the GNU C compiler on Linux OS based on Intel® 64 architecture:

```
cd interfaces/fftw3x_cdft
make libintel64 compiler=gnu mpi=openmpi INSTALL_DIR=/my/path
```

This command builds the wrapper library using the GNU gcc compiler so that the final executable can use Open MPI, and places the result, named `libfftw3x_cdft_DOUBLE.a`, into directory `/my/path`.

Building an Application

Normally, the only change needed to build your application with MPI FFTW wrappers replacing original FFTW3 library is to add Intel® oneAPI Math Kernel Library (oneMKL) and the wrapper library at the link stage (see section "Linking Your Application with Intel® oneAPI Math Kernel Library" in the *Intel® oneAPI Math Kernel Library (oneMKL) Developer Guide*).

When you are recompiling your application, add subdirectory `include\fftw` to the search path for header files to avoid FFTW3 version conflicts.

Running Examples

There are some examples that demonstrate how to use the MPI FFTW wrapper library for FFTW3. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw3x_cdft` subdirectory in the Intel® oneAPI Math Kernel Library (oneMKL) directory.

Parameters of the example makefiles are similar to the parameters of the wrapper library makefiles. Example makefiles normally build and invoke the examples. Results of running the examples are saved in subdirectory `._results` in files with extension `.res`.

For detailed information about options for the example makefile, refer to the makefile.

See Also

[Building Your Own Wrapper Library](#)

Appendix D: Code Examples

This appendix presents code examples of using some Intel® oneAPI Math Kernel Library (oneMKL) routines and functions.

Please refer to respective sections in the document for detailed descriptions of function parameters and operation.

BLAS Code Examples

Example. Using BLAS Level 1 Function

The following example illustrates a call to the BLAS Level 1 function `sdot`. This function performs a vector-vector operation of computing a scalar product of two single-precision real vectors `x` and `y`.

Parameters

<code>n</code>	Specifies the number of elements in vectors <code>x</code> and <code>y</code> .
<code>incx</code>	Specifies the increment for the elements of <code>x</code> .
<code>incy</code>	Specifies the increment for the elements of <code>y</code> .

```
#include <stdio.h>
#include <stdlib.h>

#include "mkl_example.h"

int main()
{
    MKL_INT n, incx, incy, i;
```

```

float    *x, *y;
float    res;
MKL_INT  len_x, len_y;

n = 5;
incx = 2;
incy = 1;

len_x = 1+(n-1)*abs(incx);
len_y = 1+(n-1)*abs(incy);
x = (float *)calloc( len_x, sizeof( float ) );
y = (float *)calloc( len_y, sizeof( float ) );
if( x == NULL || y == NULL ) {
    printf( "\n Can't allocate memory for arrays\n");
    return 1;
}

for (i = 0; i < n; i++) {
    x[i*abs(incx)] = 2.0;
    y[i*abs(incy)] = 1.0;
}

res = cblas_sdot(n, x, incx, y, incy);

printf("\n          SDOT = %7.3f", res);

free(x);
free(y);

return 0;
}

```

As a result of this program execution, the following line is printed:

SDOT = 10.000

Example. Using BLAS Level 1 Routine

The following example illustrates a call to the BLAS Level 1 routine `scopy`. This routine performs a vector-vector operation of copying a single-precision real vector *x* to a vector *y*.

Parameters

<i>n</i>	Specifies the number of elements in vectors <i>x</i> and <i>y</i> .
<i>incx</i>	Specifies the increment for the elements of <i>x</i> .
<i>incy</i>	Specifies the increment for the elements of <i>y</i> .

```

#include <stdio.h>
#include <stdlib.h>

#include "mkl_example.h"

int main()
{
    MKL_INT  n, incx, incy, i;
    float    *x, *y;

```



```

MKL_INT  len_x, len_y;
n = 3;
incx = 3;
incy = 1;

len_x = 10;
len_y = 10;
x = (float *)calloc( len_x, sizeof( float ) );
y = (float *)calloc( len_y, sizeof( float ) );
if( x == NULL || y == NULL ) {
    printf( "\n Can't allocate memory for arrays\n");
    return 1;
}
for (i = 0; i < 10; i++) {
    x[i] = i + 1;
}

cblas_scopy(n, x, incx, y, incy);

/*      Print output data                                */

printf("\n\n      OUTPUT DATA");
PrintVectorS(FULLPRINT, n, y, incy, "Y");

free(x);
free(y);
return 0;
}

```

As a result of this program execution, the following line is printed:

Y = 1.00000 4.00000 7.00000

Example. Using BLAS Level 2 Routine

The following example illustrates a call to the BLAS Level 2 routine `sger`. This routine performs a matrix-vector operation

$$a := \alpha x y' + a.$$

Parameters

<i>alpha</i>	Specifies a scalar <i>alpha</i> .
<i>x</i>	<i>m</i> -element vector.
<i>y</i>	<i>n</i> -element vector.
<i>a</i>	<i>m</i> -by- <i>n</i> matrix.

```

#include <stdio.h>
#include <stdlib.h>

#include "mkl_example.h"

int main()
{
    MKL_INT      m, n, lda, incx, incy, i, j;
    MKL_INT      rmaxa, cmaxa;

```

```

float      alpha;
float      *a, *x, *y;
CBLAS_LAYOUT layout;
MKL_INT    len_x, len_y;

m = 2;
n = 3;
lda = 5;
incx = 2;
incy = 1;
alpha = 0.5;

        layout = CblasRowMajor;

len_x = 10;
len_y = 10;
rmaxa = m + 1;
cmaxa = n;
a = (float *)calloc( rmaxa*cmaxa, sizeof(float) );
x = (float *)calloc( len_x, sizeof(float) );
y = (float *)calloc( len_y, sizeof(float) );
if( a == NULL || x == NULL || y == NULL ) {
    printf( "\n Can't allocate memory for arrays\n");
    return 1;
}
if( layout == CblasRowMajor )
    lda=cmaxa;
else
    lda=rmaxa;

for (i = 0; i < 10; i++) {
    x[i] = 1.0;
    y[i] = 1.0;
}

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        a[i + j*lda] = j + 1;
    }
}

cblas_sger(layout, m, n, alpha, x, incx, y, incy, a, lda);

PrintArrayS(&layout, FULLPRINT, GENERAL_MATRIX, &m, &n, a, &lda, "A");

free(a);
free(x);
free(y);

return 0;
}

```

As a result of this program execution, matrix *a* is printed as follows:

Matrix A:

```

1.50000 2.50000 3.50000
1.50000 2.50000 3.50000

```

Example. Using BLAS Level 3 Routine

The following example illustrates a call to the BLAS Level 3 routine `ssymm`. This routine performs a matrix-matrix operation

$$c := \alpha * a * b' + \beta * c.$$

Parameters

<i>alpha</i>	Specifies a scalar <i>alpha</i> .
<i>beta</i>	Specifies a scalar <i>beta</i> .
<i>a</i>	Symmetric matrix
<i>b</i>	<i>m</i> -by- <i>n</i> matrix
<i>c</i>	<i>m</i> -by- <i>n</i> matrix

```
#include <stdio.h>
#include <stdlib.h>

#include "mkl_example.h"

int main(int argc, char *argv[])
{
    MKL_INT      m, n, i, j;
    MKL_INT      lda, ldb, ldc;
    MKL_INT      rmaxa, cmaxa, rmaxb, cmAXB, rmaxc, cmAXc;
    float        alpha, beta;
    float        *a, *b, *c;
    CBLAS_LAYOUT layout;
    CBLAS_SIDE    side;
    CBLAS_UPLO    uplo;
    MKL_INT      ma, na, typeA;
    uplo = 'u';
    side = 'l';
    layout = CblasRowMajor;
    m = 3;
    n = 2;
    lda = 3;
    ldb = 3;
    ldc = 3;
    alpha = 0.5;
    beta = 2.0;

    if( side == CblasLeft ) {
        rmaxa = m + 1;
        cmaxa = m;
        ma    = m;
        na    = m;
    } else {
        rmaxa = n + 1;
        cmaxa = n;
        ma    = n;
        na    = n;
    }
    rmaxb = m + 1;
    cmAXB = n;
    rmaxc = m + 1;
    cmAXc = n;
```

```

a = (float *)calloc( rmaxa*cmaxa, sizeof(float) );
b = (float *)calloc( rmaxb*cmaxb, sizeof(float) );
c = (float *)calloc( rmaxc*cmaxc, sizeof(float) );
if ( a == NULL || b == NULL || c == NULL ) {
    printf("\n Can't allocate memory arrays");
    return 1;
}
if( layout == CblasRowMajor ) {
    lda=cmaxa;
    ldb=cmaxb;
    ldc=cmaxc;
} else {
    lda=rmaxa;
    ldb=rmaxb;
    ldc=rmaxc;
}
if (uplo == CblasUpper) typeA = UPPER_MATRIX;
else typeA = LOWER_MATRIX;
for (i = 0; i < m; i++) {
    for (j = 0; j < m; j++) {
        a[i + j*lda] = 1.0;
    }
}
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        c[i + j*ldc] = 1.0;
        b[i + j*ldb] = 2.0;
    }
}

cblas_ssymm(layout, side, uplo, m, n, alpha, a, lda,
            b, ldb, beta, c, ldc);

printf("\n\n      OUTPUT DATA");
PrintArrayS(&layout, FULLPRINT, GENERAL_MATRIX, &m, &n, c, &ldc, "C");

free(a);
free(b);
free(c);

return 0;
}

```

As a result of this program execution, matrix c is printed as follows:

Matrix C:

```

5.00000 5.00000
5.00000 5.00000
5.00000 5.00000

```

The following example illustrates a call from a C program to the Fortran version of the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

Example. Calling a Complex BLAS Level 1 Function from C

In this example, the complex dot product is returned in the structure `c`.

```
#include <stdio>
#include "mkl_blas.h"
#define N 5
void main()
{
    int n, inca = 1, incb = 1, i;
    MKL_Complex16 a[N], b[N], c;
    void zdotc();
    n = N;
    for( i = 0; i < n; i++ ){
        a[i].real = (double)i; a[i].imag = (double)i * 2.0;
        b[i].real = (double)(n - i); b[i].imag = (double)i * 2.0;
    }
    zdotc( &c, &n, a, &inca, b, &incb );
    printf( "The complex dot product is: ( %6.2f, %6.2f )\n", c.real, c.imag );
}
```

NOTE

Instead of calling BLAS directly from C programs, you might wish to use the C interface to the Basic Linear Algebra Subprograms (CBLAS) implemented in Intel® oneAPI Math Kernel Library (oneMKL). See [C Interface Conventions](#) for more information.

Fourier Transform Functions Code Examples

This section presents code examples for functions described in the “[FFT Functions](#)” and “[Cluster FFT Functions](#)” subsections in the “Fourier Transform Functions” section. The examples are grouped in subsections

- [Examples for FFT Functions](#), including [Examples of Using Multi-Threading for FFT Computation](#)
- [Examples for Cluster FFT Functions](#)
- [Auxiliary data transformations](#).

FFT Code Examples

This section presents examples of using the FFT interface functions described in “[Fourier Transform Functions](#)”.

Here are the examples of two one-dimensional computations. These examples use the default settings for all of the configuration parameters, which are specified in “[Configuration Settings](#)”.

One-dimensional In-place FFT

```
/* C example, float _Complex is defined in C9X */
#include "mkl_dfti.h"

float _Complex c2c_data[32];
float r2c_data[34];
DFTI_DESCRIPTOR_HANDLE my_desc1_handle = NULL;
DFTI_DESCRIPTOR_HANDLE my_desc2_handle = NULL;
MKL_LONG status;

/* ...put values into c2c_data[i] 0<=i<=31 */
/* ...put values into r2c_data[i] 0<=i<=31 */
```

```

status = DftiCreateDescriptor(&my_desc1_handle, DFTI_SINGLE,
                             DFTI_COMPLEX, 1, 32);
status = DftiCommitDescriptor(my_desc1_handle);
status = DftiComputeForward(my_desc1_handle, c2c_data);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is c2c_data[i] 0<=i<=31 */
status = DftiCreateDescriptor(&my_desc2_handle, DFTI_SINGLE,
                             DFTI_REAL, 1, 32);
status = DftiCommitDescriptor(my_desc2_handle);
status = DftiComputeForward(my_desc2_handle, r2c_data);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is the complex value r2c_data[i] 0<=i<=31 */
/* and is stored in CCS format*/

```

One-dimensional Out-of-place FFT

```

/* C example, float _Complex is defined in C9X */
#include "mkl_dfti.h"

float _Complex c2c_input[32];
float _Complex c2c_output[32];
float r2c_input[32];
float r2c_output[34];
DFTI_DESCRIPTOR_HANDLE my_desc1_handle = NULL;
DFTI_DESCRIPTOR_HANDLE my_desc2_handle = NULL;
MKL_LONG status;

/* ...put values into c2c_input[i] 0<=i<=31 */
/* ...put values into r2c_input[i] 0<=i<=31 */

status = DftiCreateDescriptor(&my_desc1_handle, DFTI_SINGLE,
                             DFTI_COMPLEX, 1, 32);
status = DftiSetValue(my_desc1_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor(my_desc1_handle);
status = DftiComputeForward(my_desc1_handle, c2c_input, c2c_output);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is c2c_output[i] 0<=i<=31 */
status = DftiCreateDescriptor(&my_desc2_handle, DFTI_SINGLE,
                             DFTI_REAL, 1, 32);
Status = DftiSetValue(my_desc1_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor(my_desc2_handle);
status = DftiComputeForward(my_desc2_handle, r2c_input, r2c_output);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is the complex r2c_data[i] 0<=i<=31 and is stored in CCS format*/

```

Two-dimensional FFT

```

/* C99 example */
#include "mkl_dfti.h"

/* complex data in, complex data out */
float _Complex c2c_data[32][100];
/* real data in, complex data out */
float r2c_data[34][102];

```

```

DFTI_DESCRIPTOR_HANDLE my_desc1_handle = NULL;
DFTI_DESCRIPTOR_HANDLE my_desc2_handle = NULL;
MKL_LONG status; MKL_LONG dim_sizes[2] = {32, 100};

/* ...put values into c2c_data[i][j] 0<=i<=31, 0<=j<=99 */
/* ...put values into r2c_data[i][j] 0<=i<=31, 0<=j<=99 */

status = DftiCreateDescriptor(&my_desc1_handle, DFTI_SINGLE,
                             DFTI_COMPLEX, 2, dim_sizes);
status = DftiCommitDescriptor(my_desc1_handle);
status = DftiComputeForward(my_desc1_handle, c2c_data);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is the complex value c2c_data[i][j], 0<=i<=31, 0<=j<=99 */
status = DftiCreateDescriptor(&my_desc2_handle, DFTI_SINGLE,
                             DFTI_REAL, 2, dim_sizes);
status = DftiCommitDescriptor(my_desc2_handle);
status = DftiComputeForward(my_desc2_handle, r2c_data);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is the complex r2c_data[i][j] 0<=i<=31, 0<=j<=99
   and is stored in CCS format*/

```

The following example demonstrates how you can change the default configuration settings by using the `DftiSetValue` function.

For instance, to preserve the input data after the FFT computation, the configuration of `DFTI_PLACEMENT` should be changed to "not in place" from the default choice of "in place."

The code below illustrates how this can be done:

Changing Default Settings

```

/* C99 example */
#include "mkl_dfti.h"

float _Complex x_in[32], x_out[32];
DFTI_DESCRIPTOR_HANDLE my_desc_handle = NULL;
MKL_LONG status;

/* ...put values into x_in[i] 0<=i<=31 */

status = DftiCreateDescriptor(&my_desc_handle, DFTI_SINGLE,
                             DFTI_COMPLEX, 1, 32);
status = DftiSetValue(my_desc_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor(my_desc_handle);
status = DftiComputeForward(my_desc_handle, x_in, x_out);
status = DftiFreeDescriptor(&my_desc_handle);
/* result is the complex value x_out[i], 0<=i<=31 */

```

Using Status Checking Functions

This example illustrates the use of status checking functions described in ["Fourier Transform Functions"](#).

```

/* C99 example */
#include "mkl_dfti.h"

DFTI_DESCRIPTOR_HANDLE desc = NULL;

```

```

MKL_LONG status;

/* ...descriptor creation and other code */

status = DftiCommitDescriptor(desc);
if (status && !DftiErrorClass(status, DFTI_NO_ERROR))
{
    printf('Error: %s\n', DftiErrorMessage(status));
}

```

Computing 2D FFT by One-Dimensional Transforms

Below is an example where a 20-by-40 two-dimensional FFT is computed explicitly using one-dimensional transforms.

```

/* C */
#include "mkl_dfti.h"

float _Complex data[20][40];
MKL_LONG status;
DFTI_DESCRIPTOR_HANDLE desc_handle_dim1 = NULL;
DFTI_DESCRIPTOR_HANDLE desc_handle_dim2 = NULL;

/* ...put values into data[i][j] 0<=i<=19, 0<=j<=39 */

status = DftiCreateDescriptor(&desc_handle_dim1, DFTI_SINGLE,
                             DFTI_COMPLEX, 1, 20);
status = DftiCreateDescriptor(&desc_handle_dim2, DFTI_SINGLE,
                             DFTI_COMPLEX, 1, 40);

/* perform 40 one-dimensional transforms along 1st dimension */
/* note that the 1st dimension data are not unit-stride */
MKL_LONG stride[2] = {0, 40};

status = DftiSetValue(desc_handle_dim1, DFTI_NUMBER_OF_TRANSFORMS, 40);
status = DftiSetValue(desc_handle_dim1, DFTI_INPUT_DISTANCE, 1);
status = DftiSetValue(desc_handle_dim1, DFTI_OUTPUT_DISTANCE, 1);
status = DftiSetValue(desc_handle_dim1, DFTI_INPUT_STRIDES, stride);
status = DftiSetValue(desc_handle_dim1, DFTI_OUTPUT_STRIDES, stride);
status = DftiCommitDescriptor(desc_handle_dim1);
status = DftiComputeForward(desc_handle_dim1, data);
/* perform 20 one-dimensional transforms along 2nd dimension */
/* note that the 2nd dimension is unit stride */
status = DftiSetValue(desc_handle_dim2, DFTI_NUMBER_OF_TRANSFORMS, 20);
status = DftiSetValue(desc_handle_dim2, DFTI_INPUT_DISTANCE, 40);
status = DftiSetValue(desc_handle_dim2, DFTI_OUTPUT_DISTANCE, 40);
status = DftiCommitDescriptor(desc_handle_dim2);
status = DftiComputeForward(desc_handle_dim2, data);
status = DftiFreeDescriptor(&desc_handle_dim1);
status = DftiFreeDescriptor(&desc_handle_dim2);

```

The following code illustrates real multi-dimensional transforms with CCE format storage of conjugate-even complex matrix. [Example "Three-Dimensional REAL FFT \(C Interface\)"](#) is three-dimensional out-of-place transform in C interface.

Three-Dimensional REAL FFT

```

/* C99 example */
#include "mkl_dfti.h"

float input[32][100][19];
float _Complex output[32][100][10]; /* 10 = 19/2 + 1 */
DFTI_DESCRIPTOR_HANDLE my_desc_handle = NULL;
MKL_LONG status;
MKL_LONG dim_sizes[3] = {32, 100, 19};
MKL_LONG strides_out[4] = {0, 100*10*1, 10*1, 1};

//...put values into input[i][j][k] 0<=i<=31; 0<=j<=99, 0<=k<=9

status = DftiCreateDescriptor(&my_desc_handle,
                             DFTI_SINGLE, DFTI_REAL, 3, dim_sizes);
status = DftiSetValue(my_desc_handle,
                      DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX);
status = DftiSetValue(my_desc_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiSetValue(my_desc_handle, DFTI_OUTPUT_STRIDES, strides_out);

status = DftiCommitDescriptor(my_desc_handle);
status = DftiComputeForward(my_desc_handle, input, output);
status = DftiFreeDescriptor(&my_desc_handle);
/* result is the complex output[i][j][k] 0<=i<=31; 0<=j<=99, 0<=k<=9
   and is stored in CCE format. */

```

Examples of Using OpenMP* Threading for FFT Computation

The following sample program shows how to employ internal OpenMP* threading in Intel® oneAPI Math Kernel Library (oneMKL) for FFT computation.

To specify the number of threads inside Intel® oneAPI Math Kernel Library (oneMKL), use the following settings:

set MKL_NUM_THREADS = 1 for one-threaded mode;

set MKL_NUM_THREADS = 4 for multi-threaded mode.

Using oneMKL Internal Threading Mode (C Example)

```

/* C99 example */
#include "mkl_dfti.h"

float data[200][100];
DFTI_DESCRIPTOR_HANDLE fft = NULL;
MKL_LONG dim_sizes[2] = {200, 100};

/* ...put values into data[i][j] 0<=i<=199, 0<=j<=99 */

DftiCreateDescriptor(&fft, DFTI_SINGLE, DFTI_REAL, 2, dim_sizes);
DftiCommitDescriptor(fft);
DftiComputeForward(fft, data);
DftiFreeDescriptor(&fft);

```

The following [Example “Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region”](#) and [Example “Using Parallel Mode with Multiple Descriptors Initialized in One Thread”](#) illustrate a parallel customer program with each descriptor instance used only in a single thread.

Specify the number of OpenMP threads for [Example “Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region”](#) like this:

set `MKL_NUM_THREADS = 1` for Intel® oneAPI Math Kernel Library (oneMKL) to work in the single-threaded mode (recommended);

set `OMP_NUM_THREADS = 4` for the customer program to work in the multi-threaded mode.

Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region

Note that in this example, the program can be transformed to become single-threaded at the customer level but using parallel mode within Intel® oneAPI Math Kernel Library (oneMKL). To achieve this, you must set the parameter `DFTI_NUMBER_OF_TRANSFORMS = 4` and to set the corresponding parameter `DFTI_INPUT_DISTANCE = 5000`.

```
/* C99 example */
#include "mkl_dfti.h"
#include <omp.h>
#define ARRAY_LEN(a) sizeof(a)/sizeof(a[0])

// 4 OMP threads, each does 2D FFT 50x100 points
MKL_Complex8 data[4][50][100];
int nth = ARRAY_LEN(data);
MKL_LONG dim_sizes[2] = {
    ARRAY_LEN(data[0]),
    ARRAY_LEN(data[0][0])
}; /* {50, 100} */
int th;

/* ...put values into data[i][j][k] 0<=i<=3, 0<=j<=49, 0<=k<=99 */

// assume data is initialized and do 2D FFTs
#pragma omp parallel for shared(dim_sizes, data)
for (th = 0; th < nth; ++th)
{
    DFTI_DESCRIPTOR_HANDLE myFFT = NULL;

    DftiCreateDescriptor(&myFFT, DFTI_SINGLE, DFTI_COMPLEX, 2, dim_sizes);
    DftiCommitDescriptor(myFFT);
    DftiComputeForward(myFFT, data[th]);
    DftiFreeDescriptor(&myFFT);
}
```

Specify the number of OpenMP threads for [Example “Using Parallel Mode with Multiple Descriptors Initialized in One Thread”](#) like this:

set `MKL_NUM_THREADS = 1` for Intel® oneAPI Math Kernel Library (oneMKL) to work in the single-threaded mode (obligatory);

set `OMP_NUM_THREADS = 4` for the customer program to work in the multi-threaded mode.

Using Parallel Mode with Multiple Descriptors Initialized in One Thread

```
/* C99 example */
#include "mkl_dfti.h"
#include <omp.h>
#define ARRAY_LEN(a) sizeof(a)/sizeof(a[0])

// 4 OMP threads, each does 2D FFT 50x100 points
MKL_Complex8 data[4][50][100];
int nth = ARRAY_LEN(data);
MKL_LONG dim_sizes[2] = {
```

```

    ARRAY_LEN(data[0]),
    ARRAY_LEN(data[0][0])
}; /* {50, 100} */
DFTI_DESCRIPTOR_HANDLE FFT[ARRAY_LEN(data)];
int th;

/* ...put values into data[i][j][k] 0<=i<=3, 0<=j<=49, 0<=k<=99 */

for (th = 0; th < nth; ++th)
    DftiCreateDescriptor(&FFT[th], DFTI_SINGLE, DFTI_COMPLEX, 2, dim_sizes);
for (th = 0; th < nth; ++th)
    DftiCommitDescriptor(FFT[th]);

// assume data is initialized and do 2D FFTs
#pragma omp parallel for shared(FFT, data)
for (th = 0; th < nth; ++th)
    DftiComputeForward(FFT[th], data[th]);

for (th = 0; th < nth; ++th)
    DftiFreeDescriptor(&FFT[th]);

```

The following [Example “Using Parallel Mode with a Common Descriptor”](#) illustrates a parallel customer program with a common descriptor used in several threads.

Using Parallel Mode with a Common Descriptor

```

#include "mkl_dfti.h"
#include <omp.h>
#define ARRAY_LEN(a) sizeof(a)/sizeof(a[0])

// 4 OMP threads, each does 2D FFT 50x100 points
MKL_Complex8 data[4][50][100];
int nth = ARRAY_LEN(data);
MKL_LONG len[2] = {ARRAY_LEN(data[0]), ARRAY_LEN(data[0][0])};
DFTI_DESCRIPTOR_HANDLE FFT;
int th;

/* ...put values into data[i][j][k] 0<=i<=3, 0<=j<=49, 0<=k<=99 */

DftiCreateDescriptor(&FFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len);
DftiCommitDescriptor(FFT);

// assume data is initialized and do 2D FFTs
#pragma omp parallel for shared(FFT, data)
for (th = 0; th < nth; ++th)
    DftiComputeForward(FFT, data[th]);
DftiFreeDescriptor(&FFT);

```

Examples for Cluster FFT Functions

The following C example computes a 2-dimensional out-of-place FFT using the cluster FFT interface:

2D Out-of-place Cluster FFT Computation

```

/* C99 example */
#include "mpi.h"
#include "mkl_cdft.h"

DFTI_DESCRIPTOR_DM_HANDLE desc = NULL;

```

```

MKL_LONG v, i, j, n, s;
Complex *in, *out;
MKL_LONG dim_sizes[2] = {nx, ny};

MPI_Init(...);

/* Create descriptor for 2D FFT */
DftiCreateDescriptorDM(MPI_COMM_WORLD,
                      &desc, DFTI_DOUBLE, DFTI_COMPLEX, 2, dim_sizes);
/* Ask necessary length of in and out arrays and allocate memory */
DftiGetValueDM(desc, CDFT_LOCAL_SIZE, &v);
in = (Complex*) malloc(v*sizeof(Complex));
out = (Complex*) malloc(v*sizeof(Complex));
/* Fill local array with initial data. Current process performs n rows,
   0 row of in corresponds to s row of virtual global array */
DftiGetValueDM(desc, CDFT_LOCAL_NX, &n);
DftiGetValueDM(desc, CDFT_LOCAL_X_START, &s);
/* Virtual global array globalIN is defined by function f as
   globalIN[i*ny+j]=f(i,j) */
for(i = 0; i < n; ++i)
    for(j = 0; j < ny; ++j) in[i*ny+j] = f(i+s,j);
/* Set that we want out-of-place transform (default is DFTI_INPLACE) */
DftiSetValueDM(desc, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
/* Commit descriptor, calculate FFT, free descriptor */
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc, in, out);
/* Virtual global array globalOUT is defined by function g as
   globalOUT[i*ny+j]=g(i,j)   Now out contains result of FFT. out[i*ny+j]=g(i+s,j) */
DftiFreeDescriptorDM(&desc);
free(in);
free(out);
MPI_Finalize();

```

1D In-place Cluster FFT Computations

The C example below illustrates one-dimensional in-place cluster FFT computations effected with a user-defined workspace:

```

/* C99 example */
#include "mpi.h"
#include "mkl_cdft.h"

DFTI_DESCRIPTOR_DM_HANDLE desc = NULL;
MKL_LONG N, v, i, n_out, s_out;
Complex *in, *work;

MPI_Init(...);
/* Create descriptor for 1D FFT */
DftiCreateDescriptorDM(MPI_COMM_WORLD, &desc, DFTI_DOUBLE, DFTI_COMPLEX, 1, N);
/* Ask necessary length of array and workspace and allocate memory */
DftiGetValueDM(desc, CDFT_LOCAL_SIZE, &v);
in = (Complex*) malloc(v*sizeof(Complex));
work = (Complex*) malloc(v*sizeof(Complex));
/* Fill local array with initial data. Local array has n elements,
   0 element of in corresponds to s element of virtual global array */
DftiGetValueDM(desc, CDFT_LOCAL_NX, &n);
DftiGetValueDM(desc, CDFT_LOCAL_X_START, &s);
/* Set work array as a workspace */

```

```

DftiSetValueDM(desc, CDFT_WORKSPACE, work);
/* Virtual global array globalIN is defined by function f as globalIN[i]=f(i) */
for(i = 0; i < n; ++i) in[i] = f(i+s);
/* Commit descriptor, calculate FFT, free descriptor */
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc,in);
DftiGetValueDM(desc, CDFT_LOCAL_OUT_NX, &n_out);
DftiGetValueDM(desc, CDFT_LOCAL_OUT_X_START, &s_out);
/* Virtual global array globalOUT is defined by function g as globalOUT[i]=g(i)
   Now in contains result of FFT. Local array has n_out elements,
   0 element of in corresponds to s_out element of virtual global array.
   in[i]==g(i+s_out) */
DftiFreeDescriptorDM(&desc);
free(in);
free(work);
MPI_Finalize();

```

Auxiliary Data Transformations

This section presents C examples for conversion from the Cartesian to polar representation of complex data and vice versa.

Conversion from Cartesian to polar representation of complex data

```

// Cartesian->polar conversion of complex data
// Cartesian representation: z = re + I*im
// Polar representation: z = r * exp( I*phi )
#include <mkl_vml.h>

void
variant1_Cartesian2Polar(int n,const double *re,const double *im,
                        double *r,double *phi)
{
    vdHypot(n,re,im,r);          // compute radii r[]
    vdAtan2(n,im,re,phi);        // compute phases phi[]
}

void
variant2_Cartesian2Polar(int n,const MKL_Complex16 *z,double *r,double *phi,
                        double *temp_re,double *temp_im)
{
    vzAbs(n,z,r);                // compute radii r[]
    vdPackI(n, (double*)z + 0, 2, temp_re);
    vdPackI(n, (double*)z + 1, 2, temp_im);
    vdAtan2(n,temp_im,temp_re,phi); // compute phases phi[]
}

```

Conversion from polar to Cartesian representation of complex data

```

// Polar->Cartesian conversion of complex data.
// Polar representation: z = r * exp( I*phi )
// Cartesian representation: z = re + I*im
#include <mkl_vml.h>

void

```

```

variant1_Polar2Cartesian(int n,const double *r,const double *phi,
                        double *re,double *im)
{
    vdSinCos(n,phi,im,re);    // compute direction, i.e. z[]/abs(z[])
    vdMul(n,r,re,re);        // scale real part
    vdMul(n,r,im,im);        // scale imaginary part
}

void
variant2_Polar2Cartesian(int n,const double *r,const double *phi,
                        MKL_Complex16 *z,
                        double *temp_re,double *temp_im)
{
    vdSinCos(n,phi,temp_im,temp_re); // compute direction, i.e. z[]/abs(z[])
    vdMul(n,r,temp_im,temp_im); // scale imaginary part
    vdMul(n,r,temp_re,temp_re); // scale real part
    vdUnpackI(n,temp_re,(double*)z + 0, 2); // fill in result.re
    vdUnpackI(n,temp_im,(double*)z + 1, 2); // fill in result.im
}

```

Appendix F: oneMKL Functionality

This appendix provides an overview of the Intel® oneAPI Math Kernel Library (oneMKL) functionality on the different devices.

BLAS Functionality

C

Functionality	CPU	OpenMP Offload Intel GPU
Level 1 BLAS (standard)	All	All
Level 2 BLAS (standard)	All	All
Level 3 BLAS (standard)	All	All
BLAS Extensions and Specializations	{AXPY,GEMM,TRSM}_BATCH (group and strided)	{AXPY,GEMM,TRSM}_BATCH (group and strided)
	GEMMT, AXPBY, GEMM3M	GEMMT
	Integer GEMM (s8u8)	N/A
	Bfloat16 GEMM	N/A
	JIT GEMM API	N/A
	PACK GEMM API	N/A
	COMPACT GEMM API	N/A

Transposition Functionality

Functionality	CPU	OpenMP Offload Intel GPU
In-place dense matrix transpose	Yes	No
Out-of-place dense matrix transpose	Yes	No
In-place dense matrix add	Yes	No
Out-of-place dense matrix copy	Yes	No

LAPACK Functionality

NOTE All of the DPC++ LAPACK computational routines have a corresponding `*_scratchpad_size` function for calculating the required amount of scratchpad space.

LU Factorization Routines

Functionality	CPU	OpenMP Offload Intel GPU
getrf	Yes	Yes
getrs	Yes	Yes
getri	Yes	Yes

Cholesky Factorization Routines

Functionality	CPU	OpenMP Offload Intel GPU
potrf	Yes	Yes
potrs	Yes	Yes
potri	Yes	Yes

Orthogonal Factorization Routines

Functionality	CPU	OpenMP Offload Intel GPU
geqrf	Yes	Yes
{or,un}gqr	Yes	Yes
{or,un}mqr	Yes	Yes
gerqf	Yes	No
{or,un}mrq	Yes	No

Other Linear Equation Routines

Functionality	CPU	OpenMP Offload Intel GPU
trtrs	Yes	Yes
{sy,he}trf	Yes	No

Symmetric Eigenvalue Routines

Functionality	CPU	OpenMP Offload Intel GPU
{sy,he}ev	Yes	Yes

Functionality	CPU	OpenMP Offload Intel GPU
{sy,he}evd	Yes	Yes
{sy,he}evx	Yes	Yes
{sy,he}trd	Yes	Yes
{or,un}gtr	Yes	No
{or,un}mtr	Yes	No
steqr	Yes	Yes

Generalized Symmetric Eigenvalue Routines

Functionality	CPU	OpenMP Offload Intel GPU
{sy,he}gvd	Yes	Yes
{sy,he}gvx	Yes	Yes

Singular Value Routines

Functionality	CPU	OpenMP Offload Intel GPU
gesvd	Yes	Yes
gebrd	Yes	Yes
{or,un}gbr	Yes	No

Batched LAPACK Routines

Functionality	CPU	OpenMP Offload Intel GPU (ILP64 Interface)
getrf_batch	Strided	Strided
getrfnp_batch	Strided	Strided
getrs_batch	Strided	Strided
getrsnp_batch	Strided	Strided
getri_batch	No	No
potrf_batch	No	No
potrs_batch	No	No
geqrf_batch	No	No
{or,un}gqr_batch	No	No

Other LAPACK Routines

CPU	OpenMP Offload Intel GPU
Yes	No

DFT Functionality

DFTI Interfaces

Functionality	CPU	OpenMP Offload Intel GPU
Complex-to-Complex FFT transformations	Yes, 1D through 7D	Yes, 1D through 3D

Functionality	CPU	OpenMP Offload Intel GPU
Real-to-Complex FFT transformations	Yes, 1D through 7D	Yes, 1D through 3D

FFTW3 Interfaces

Functionality	CPU	OpenMP Offload Intel GPU
Complex-to-Complex FFT transformations	Yes, 1D through 7D	Yes, 1D through 3D
Real-to-Complex FFT transformations	Yes, 1D through 7D	No

FFTW2 Interfaces

Functionality	CPU	OpenMP Offload Intel GPU
Complex-to-Complex FFT transformations	Yes, 1D through 7D	No
Real-to-Complex FFT transformations	Yes, 1D through 7D	No

Sparse BLAS Functionality

In the following table for functionality, sm = sparse matrix, dm = dense matrix, sv = sparse vector, dv = dense vector, sc = scalar.

In the following table for operations, dense vectors = x, y, sparse vectors = w,v, dense matrices = X,Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Level 1

Functionality	Operations	CPU	OpenMP Offload Intel GPU
Sparse Vector - Dense Vector addition (AXPY)	$y \leftarrow \alpha * w + y$	Yes	No
Sparse Vector - Sparse Vector Dot product (SPDOT) (sv.sv -> sc)	$d \leftarrow \text{dot}(w,v)$	N/A	N/A
	$\text{dot}(w,v) = \sum(w_i * v_i)$	No	No
	$\text{dot}(w,v) = \sum(\text{conj}(w_i) * v_i)$	No	No
Sparse Vector - Dense Vector Dot product (SPDOT) (sv.dv -> sc)	$d \leftarrow \text{dot}(w,x)$	N/A	N/A
	$\text{dot}(w,v) = \sum(w_i * v_i)$	Yes	No
	$\text{dot}(w,v) = \sum(\text{conj}(w_i) * v_i)$	Yes	No
Dense Vector - Sparse Vector Conversion (sv <-> dv)	—	N/A	N/A
	$x = \text{scatter}(w)$	Yes	No
	$w = \text{gather}(x, \text{windx})$	Yes	No

In the following table for functionality, sm = sparse matrix, dm = dense matrix, sv = sparse vector, dv = dense vector, sc = scalar.

In the following table for operations, dense vectors = x, y, sparse vectors = w,v, dense matrices = X,Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Level 2

Functionality	Operations	CPU	OpenMP Offload Intel GPU
General Matrix-Vector multiplication (GEMV) (sm*dv->dv)	$y \leftarrow \beta * y + \alpha * \text{op}(A) * x$	N/A	N/A
	$\text{op}(A) = A$	Yes	No
	$\text{op}(A) = A^T$	Yes	No
	$\text{op}(A) = A^H$	Yes	No
Symmetric Matrix-Vector multiplication (SYMV) (sm*dv->dv)	$y \leftarrow \beta * y + \alpha * \text{op}(A) * x$	N/A	N/A
	$\text{op}(A) = A$	Yes	No
	$\text{op}(A) = A^T$	Yes	No
	$\text{op}(A) = A^H$	Yes	No
Triangular Matrix-Vector multiplication (TRMV) (sm*dv->dv)	$y \leftarrow \beta * y + \alpha * \text{op}(A) * x$	N/A	N/A
	$\text{op}(A) = A$	Yes	No
	$\text{op}(A) = A^T$	Yes	No
	$\text{op}(A) = A^H$	Yes	No
General Matrix-Vector mult with dot product (GEMV DOT) (sm*dv -> dv, dv.dv->sc)	$y \leftarrow \beta * y + \alpha * \text{op}(A) * x, d = \text{dot}(x, y)$	N/A	N/A
	$\text{op}(A) = A$	Yes	No
	$\text{op}(A) = A^T$	Yes	No
	$\text{op}(A) = A^H$	Yes	No
Triangular Solve (TRSV) (inv(sm)*dv -> dv)	solve for y, $\text{op}(A) * y = \alpha * x$	N/A	N/A
	$\text{op}(A) = A$	Yes	No
	$\text{op}(A) = A^T$	Yes	No
	$\text{op}(A) = A^H$	Yes	No

In the following table for functionality, sm = sparse matrix, dm = dense matrix, sv = sparse vector, dv = dense vector, sc = scalar.

In the following table for operations, dense vectors = x, y, sparse vectors = w, v, dense matrices = X, Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Level 3

Functionality	Operations	CPU	OpenMP Offload Intel GPU
General Sparse Matrix - Dense Matrix Multiplication (GEMM) (sm*dm->dm)	$Y \leftarrow \alpha * \text{op}(A) * \text{op}(X) + \beta * Y$	N/A	N/A
	$\text{op}(A) = A, \text{op}(X) = X$	Yes	No
	$\text{op}(A) = A^T, \text{op}(X) = X$	Yes	No
	$\text{op}(A) = A^H, \text{op}(X) = X$	Yes	No

Functionality	Operations	CPU	OpenMP Offload Intel GPU
General Dense Matrix - Sparse Matrix Multiplication (GEMM) (dm*sm->dm)	$\text{op}(A) = A, \text{op}(X) = X^T$	No	No
	$\text{op}(A) = A^T, \text{op}(X) = X^T$	No	No
	$\text{op}(A) = A, \text{op}(X) = X^H$	No	No
	$\text{op}(A) = A^H$	No	No
	$\text{op}(A) = A^T, \text{op}(X) = X^H$	No	No
	$\text{op}(A) = A^H, \text{op}(X) = X^H$	No	No
	$Y \leftarrow \alpha * \text{op}(X) * \text{op}(A) + \beta * Y$	N/A	N/A
	$\text{op}(X) = X, \text{op}(A) = A$	No	No
	$\text{op}(X) = X^H, \text{op}(A) = A$	No	No
	$\text{op}(X) = X^H, \text{op}(A) = A$	No	No
	$\text{op}(X) = X, \text{op}(A) = A^H$	No	No
	$\text{op}(X) = X^H, \text{op}(A) = A^H$	No	No
	$\text{op}(X) = X^H, \text{op}(A) = A^H$	No	No
	$\text{op}(X) = X, \text{op}(A) = A^H$	No	No
	$\text{op}(X) = X^H, \text{op}(A) = A^H$	No	No
General Sparse Matrix - Sparse Matrix Multiplication (GEMM) (sm*sm->sm)	$C \leftarrow \alpha * \text{op}(A) * \text{op}(B) + \beta * C$	N/A	N/A
	$\text{op}(A) = A, \text{op}(B) = B$	Yes	No
	$\text{op}(A) = A^T, \text{op}(B) = B$	Yes	No
	$\text{op}(A) = A^H, \text{op}(B) = B$	Yes	No
	$\text{op}(A) = A, \text{op}(B) = B^T$	Yes	No
	$\text{op}(A) = A^T, \text{op}(B) = B^T$	Yes	No
	$\text{op}(A) = A^H, \text{op}(B) = B^T$	Yes	No
	$\text{op}(A) = A, \text{op}(B) = B^H$	Yes	No
	$\text{op}(A) = A^T, \text{op}(B) = B^H$	Yes	No
General Sparse Matrix - Sparse Matrix Multiplication (GEMM) (sm*sm->dm)	$Y \leftarrow \alpha * \text{op}(A) * \text{op}(B) + \beta * Y$	N/A	N/A
	$\text{op}(A) = A, \text{op}(B) = B$	Yes	No
	$\text{op}(A) = A^T, \text{op}(B) = B$	Yes	No
	$\text{op}(A) = A^H, \text{op}(B) = B$	Yes	No
	$\text{op}(A) = A, \text{op}(B) = B^T$	No	No

Functionality	Operations	CPU	OpenMP Offload Intel GPU
	$\text{op}(A)=A^T, \text{op}(B)=B^T$	No	No
	$\text{op}(A)=A^H, \text{op}(B)=B^T$	No	No
	$\text{op}(A)=A, \text{op}(B)=B^H$	No	No
	$\text{op}(A)=A^T, \text{op}(B)=B^H$	No	No
	$\text{op}(A)=A^H, \text{op}(B)=B^H$	No	No
Symmetric Rank-K update (SYRK) (sm*sm->sm)	$C \leftarrow \text{op}(A)*\text{op}(A)^H$	N/A	N/A
	$\text{op}(A)=A$	Yes	No
	$\text{op}(A)=A^T$	Yes	No
	$\text{op}(A)=A^H$	Yes	No
Symmetric Rank-K update (SYRK) (sm*sm->dm)	$Y \leftarrow \text{op}(A)*\text{op}(A)^H$	N/A	N/A
	$\text{op}(A)=A$	Yes	No
	$\text{op}(A)=A^T$	Yes	No
	$\text{op}(A)=A^H$	Yes	No
Symmetric Triple Product (SYPR) (op(sm)*sm*sm -> sm)	$C \leftarrow \text{op}(A)*B*\text{op}(A)^H$	N/A	N/A
	$\text{op}(A)=A$	Yes	No
	$\text{op}(A)=A^T$	Yes	No
	$\text{op}(A)=A^H$	Yes	No
Triangular Solve (TRSM) (inv(sm)*dm -> dm)	solve for Y, $\text{op}(A)*Y = \text{alpha}*X$	N/A	N/A
	$\text{op}(A)=A$	Yes	No
	$\text{op}(A)=A^T$	Yes	No
	$\text{op}(A)=A^H$	Yes	No

In the following table for functionality, sm = sparse matrix, dm = dense matrix, sv = sparse vector, dv = dense vector, sc = scalar.

In the following table for operations, dense vectors = x, y, sparse vectors = w,v, dense matrices = X,Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Other

Functionality	Operations	CPU	OpenMP Offload Intel GPU
Symmetric Gauss-Seidel Preconditioner (SYMGS) (update $A*x=b, A=L+D+U$)	$x0 \leftarrow x*\text{alpha}; (L+D)*x1=b-U*x0; (U+D)*x=b-L*x1$	Yes	No
Symmetric Gauss-Seidel Preconditioner with Matrix-Vector product (SYMGS_MV) (update $A*x=b, A=L+D+U$)	$x0 \leftarrow x*\text{alpha}; (L+D)*x1=b-U*x0; (U+D)*x=b-L*x1; y=A*x$	Yes	No
LU Smoother (LU_SMOOTHER) (update $A*x=b, A=L+D+U, E \sim \text{inv}(D)$)	$r=b-A*x; (L+D)*E*(U+D)*dx=r; y=x+dr$	Yes	No

Functionality	Operations	CPU	OpenMP Offload Intel GPU
Sparse Matrix Add (ADD)	$C \leftarrow \alpha * \text{op}(A) + B$	Yes	No
	$\text{op}(A) = A^T$	Yes	No
	$\text{op}(A) = A^H$	Yes	No

In the following table for operations, dense vectors = x, y, sparse vectors = w, v, dense matrices = X, Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Helper Functions

Functionality	Operations	CPU	OpenMP Offload Intel GPU
Sort Indices of Matrix (ORDER)	N/A	Yes	No
Transpose of Sparse Matrix (TRANPOSE)	$A \leftarrow \text{op}(A)$ with op=trans or conjtrans	N/A	N/A
	transpose CSR/CSC matrix	Yes	No
	transpose BSR matrix	Yes	No
Sparse Matrix Format Converter (CONVERT)	N/A	Yes	No
Dense to Sparse Matrix Format Converter (CONVERT)	N/A	Yes	No
Copy Matrix Handle (COPY)	N/A	Yes	No
Create CSR Matrix Handle	N/A	Yes	No
Create CSC Matrix Handle	N/A	Yes	No
Create COO Matrix Handle	N/A	Yes	No
Create BSR Matrix Handle	N/A	Yes	No
Export CSR Matrix	Allows access to internal data in the CSR Matrix handle	Yes	No
Export CSC Matrix	Allows access to internal data in the CSC Matrix handle	Yes	No
Export COO Matrix	Allows access to internal data in the COO Matrix handle	Yes	No
Export BSR Matrix	Allows access to internal data in the BSR Matrix handle	Yes	No
Set Value in Matrix	N/A	Yes	No

In the following table for functionality, sm = sparse matrix, dm = dense matrix, sv = sparse vector, dv = dense vector, sc = scalar.

In the following table for operations, dense vectors = x, y, sparse vectors = w, v, dense matrices = X, Y, sparse matrices = A, B, C, and scalars = alpha, beta, d.

Optimize Stages

Functionality	Operations	CPU	OpenMP Offload Intel GPU
add MEMORY hint and optimize	Chooses to allow larger memory requiring optimizations or not.	Yes	No

Functionality	Operations	CPU	OpenMP Offload Intel GPU
Add GEMV hint and optimize	N/A	Yes	No
Add SYMV hint and optimize	N/A	Yes	No
Add TRMV hint and optimize	N/A	Yes	No
add TRSV hint and optimize	N/A	Yes	No
add GEMM hint and optimize	N/A	Yes	No
add TRSM hint and optimize	N/A	Yes	No
add DOTMV hint and optimize	N/A	Yes	No
add SYMGS hint and optimize	N/A	Yes	No
add SYMGS_MV hint and optimize	N/A	Yes	No
add LU_SMOOTHER hint and optimize	N/A	Yes	No

Sparse Solvers Functionality

Functionality	CPU	OpenMP Offload Intel GPU
Sparse Cholesky Factorization	Yes	No
Sparse LU Factorization	Yes	No
Sparse QR factorization	Yes	No
Hermitian/Symmetric Eigensolver on intervals for Sparse Matrices	Yes	No
Extremal Eigensolvers for Sparse Matrix	Yes	No
Poisson Solver	Yes	No
Trust Region Solver	Yes	No

Random Number Generators Functionality

Engines

Functionality	CPU	OpenMP Offload Intel GPU
MRG32K3A	Yes	Yes
MT2203	Yes	Yes
MT19937	Yes	Yes
PHILOX4X32X10	Yes	Yes
SOBOL	Yes	Yes
ARS5	Yes	No
MCG59	Yes	Yes
NIEDERR	Yes	No

Functionality	CPU	OpenMP Offload Intel GPU
MCG31	Yes	Yes
WH	Yes	No
SFMT19937	Yes	No
R250	Yes	No
NONDETERM	Yes	No
DABSTRACT	Yes	No
SABSTRACT	Yes	No
IABSTRACT	Yes	No

Distributions

Functionality	CPU	OpenMP Offload Intel GPU
Uniform (single/double/integer)	Yes	Yes
UniformBits32	Yes	Yes
UniformBits64		
Lognormal (single/double)	Yes	Yes
Gaussian (single/double)	Yes	Yes
Poisson	Yes	Yes
UniformBits	Yes	Yes
Bernoulli	Yes	Yes
Beta (single/double)	Yes	No
Binomial	Yes	No
ChiSquare (single/double)	Yes	No
Exponential (single/double)	Yes	Yes
Gamma (single/double)	Yes	No
Geometric	Yes	Yes
Gumbel (single/double)	Yes	Yes
Hyper Geometric	Yes	No
Laplace (single/double)	Yes	Yes
Multinomial	Yes	No
Negative Binomial	Yes	No
PoissonV	Yes	No
Rayleigh (single/double)	Yes	Yes
Weibull (single/double)	Yes	Yes
Cauchy (single/double)	Yes	Yes

Functionality	CPU	OpenMP Offload Intel GPU
GaussianMV (single/double)	Yes	Yes

Vector Math Functionality

Functionality	CPU	OpenMP Offload Intel GPU
Vector Math Functions, Single Precision	Yes	Yes*
Vector Math Functions, Double Precision	Yes	Yes*
Vector Math Functions, Single Precision Complex	Yes	No
Vector Math Functions, Double Precision Complex	Yes	No

OpenMP offload to the GPU is implemented in the Linux OS, but not in the Windows* OS. The Windows OS implementation will be available in a future release.

Data Fitting Functionality

Splines, Spline Type

Functionality	CPU	OpenMP Offload Intel GPU
default (linear/quadratic/cubic)	Yes	No
subbotin	Yes	No
natural	Yes	No
hermite	Yes	No
akima	Yes	No
bessel	Yes	No
hyman	Yes	No
lookup interpolant	Yes	No
cr stepwise const interpolant	Yes	No
cl stepwise const interpolant	Yes	No

Computation Routines

Functionality	CPU	OpenMP Offload Intel GPU
Construct1D	Yes	No
Interpolate1D/Interpolate1DEx	Yes	No
Integrate1D/Integrate1DEx	Yes	No
SearchCells1D/SearchCells1DEx	Yes	No
InterpolationCallBack	Yes	No
IntegrateCallBack	Yes	No

Functionality	CPU	OpenMP Offload Intel GPU
SearchCellsCallBack	Yes	No

Summary Statistics Functionality

Functionality	CPU	OpenMP Offload Intel GPU
min	Yes	No
max	Yes	No
raw sum	Yes	No
2nd order raw sum		
3rd order raw sum		
4th order raw sum		
2nd order central sum	Yes	No
3rd order central sum		
4th order central sum		
mean	Yes	No
2nd order raw moment		
3rd order raw moment		
4th order raw moment		
2nd order central moment	Yes	No
3rd order central moment		
4th order central moment		
kurtosis	Yes	No
skewness	Yes	No
variation coefficient	Yes	No
covariance matrix	Yes	No
correlation matrix	Yes	No
cross-product matrix	Yes	No
pooled covariance matrix	Yes	No
pooled mean	Yes	No
group covariance matrix	Yes	No
group mean	Yes	No
quantiles	Yes	No
order statistics	Yes	No
robust covariance matrix	Yes	No
ouliers detection	Yes	No

Functionality	CPU	OpenMP Offload Intel GPU
partial covariance matrix	Yes	No
partial covariance matrix	Yes	No
missing values	Yes	No
parameterized correlation matrix	Yes	No
stream quantiles	Yes	No
mean absolute deviation	Yes	No
median absolute deviation	Yes	No
sorted observations	Yes	No

Bibliography

For more information about the BLAS, Sparse BLAS, LAPACK, ScaLAPACK, Sparse Solver, Extended Eigensolver, VM, VS, FFT, and Non-Linear Optimization Solvers functionality, refer to the following publications:

- **BLAS Level 1**

C. Lawson, R. Hanson, D. Kincaid, and F. Krough. *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, Vol.5, No.3 (September 1979) 308-325.

- **BLAS Level 2**

J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.14, No.1 (March 1988) 1-32.

- **BLAS Level 3**

J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software (December 1989).

- **Sparse BLAS**

D. Dodson, R. Grimes, and J. Lewis. *Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Math Software, Vol.17, No.2 (June 1991).

D. Dodson, R. Grimes, and J. Lewis. *Algorithm 692: Model Implementation and Test Package for the Sparse Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.17, No.2 (June 1991).

[Duff86] I.S.Duff, A.M.Erisman, and J.K.Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, UK, 1986.

[CXML01] *Compaq Extended Math Library*. Reference Guide, Oct.2001.

[Rem05] K.Remington. *A NIST FORTRAN Sparse Blas User's Guide*. (available on <http://math.nist.gov/~KRemington/fspblas/>)

[Saad94] Y.Saad. *SPARSKIT: A Basic Tool-kit for Sparse Matrix Computation*. Version 2, 1994.(<http://www.cs.umn.edu/~saad>)

[Saad96] Y.Saad. *Iterative Methods for Linear Systems*. PWS Publishing, Boston, 1996.

- **LAPACK**

[AndaPark94] A. A. Anda and H. Park. *Fast plane rotations with dynamic scaling*, SIAM J. matrix Anal. Appl., Vol. 15 (1994), pp. 162-174.

- [Baudin12] M. Baudin, R. Smith. *A Robust Complex Division in Scilab*, available from <http://www.arxiv.org>, arXiv:1210.4539v2 (2012).
- [Bischof00] C. H. Bischof, B. Lang, and X. Sun. *Algorithm 807: The SBR toolbox-software for successive band reduction*, ACM Transactions on Mathematical Software, Vol. 26, No. 4, pages 602-616, December 2000.
- [Demmel92] J. Demmel and K. Veselic. *Jacobi's method is more accurate than QR*, SIAM J. Matrix Anal. Appl. 13(1992):1204-1246.
- [Demmel12] J. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou. *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM Journal on Scientific Computing, Vol. 34, No 1, 2012.
- [deRijk98] P. P. M. De Rijk. *A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer*, SIAM J. Sci. Stat. Comp., Vol. 10 (1998), pp. 359-371.
- [Dhillon04] I. Dhillon, B. Parlett. *Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices*, Linear Algebra and its Applications, 387(1), pp. 1-28, August 2004.
- [Dhillon04-02] I. Dhillon, B. Parlett. *Orthogonal Eigenvectors and * Relative Gaps*, SIAM Journal on Matrix Analysis and Applications, Vol. 25, 2004. (Also LAPACK Working Note 154.)
- [Dhillon97] I. Dhillon. *A new $O(n^2)$ algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem*, Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.
- [Drmac08-1] Z. Drmac and K. Veselic. *New fast and accurate Jacobi SVD algorithm I*, SIAM J. Matrix Anal. Appl. Vol. 35, No. 2 (2008), pp. 1322-1342. LAPACK Working note 169.
- [Drmac08-2] Z. Drmac and K. Veselic. *New fast and accurate Jacobi SVD algorithm II*, SIAM J. Matrix Anal. Appl. Vol. 35, No. 2 (2008), pp. 1343-1362. LAPACK Working note 170.
- [Drmac08-3] Z. Drmac and K. Bujanovic. *On the failure of rank-revealing QR factorization software - a case study*, ACM Trans. Math. Softw. Vol. 35, No 2 (2008), pp. 1-28. LAPACK Working note 176.
- [Drmac08-4] Z. Drmac. *Implementation of Jacobi rotations for accurate singular value computation in floating point arithmetic*, SIAM J. Sci. Comp., Vol. 18 (1997), pp. 1200-1222.
- [Elmroth00] E. Elmroth and F. Gustavson. *Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance*, IBM J. Research & Development, Vol. 44, No. 4, 2000, pp 605-624.
- [Golub96] G. Golub and C. Van Loan. *Matrix Computations*, Johns Hopkins University Press, Baltimore, third edition, 1996.
- [LUG] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*, Third Edition, Society for Industrial and Applied Mathematics (SIAM), 1999.
- [Kahan66] W. Kahan. *Accurate Eigenvalues of a Symmetric Tridiagonal Matrix*, Report CS41, Computer Science Dept., Stanford University, July 21, 1966.

- [Marques06] O.Marques, E.J.Riedy, and Ch.Voemel. *Benefits of IEEE-754 Features in Modern Symmetric Tridiagonal Eigensolvers*, SIAM Journal on Scientific Computing, Vol.28, No.5, 2006. (Tech report version in LAPACK Working Note 172 with the same title.)
- [Sutton09] Brian D. Sutton. *Computing the complete CS decomposition*, Numer. Algorithms, 50(1):33-65, 2009.
- **ScaLAPACK**

[SLUG] L. Blackford, J. Choi, A.Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K.Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics (SIAM), 1997.
 - **Sparse Solver**

[Duff99] I. S. Duff and J. Koster. *The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices*. SIAM J. Matrix Analysis and Applications, 20(4):889-901, 1999.

[Dong95] J. Dongarra, V.Eijkhout, A.Kalhan. *Reverse Communication Interface for Linear Algebra Templates for Iterative Methods*. UT-CS-95-291, May 1995. <http://www.netlib.org/lapack/lawnspdf/lawn99.pdf>

[Li99] X.S. Li and J.W. Demmel. *A Scalable Sparse Direct Solver Using Static Pivoting*. In Proceeding of the 9th SIAM conference on Parallel Processing for Scientific Computing, San Antonio, Texas, March 22-34,1999.

[Liu85] J.W.H. Liu. *Modification of the Minimum-Degree algorithm by multiple elimination*. ACM Transactions on Mathematical Software, 11(2):141-153, 1985.

[Menon98] R. Menon L. Dagnum. *OpenMP: An Industry-Standard API for Shared-Memory Programming*. IEEE Computational Science & Engineering, 1:46-55, 1998. <http://www.openmp.org>.

[Saad03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd edition, SIAM, Philadelphia, PA, 2003.

[Schenk00] O. Schenk. *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*. PhD thesis, ETH Zurich, 2000.

[Schenk00-2] O. Schenk, K. Gartner, and W. Fichtner. *Efficient Sparse LU Factorization with Left-right Looking Strategy on Shared Memory Multiprocessors*. BIT, 40(1):158-176, 2000.

[Schenk01] O. Schenk and K. Gartner. *Sparse Factorization with Two-Level Scheduling in PARDISO*. In Proceeding of the 10th SIAM conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, March 12-14, 2001.

[Schenk02] O. Schenk and K. Gartner. *Two-level scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems*. Parallel Computing, 28:187-197, 2002.

[Schenk03] O. Schenk and K. Gartner. *Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO*. Journal of Future Generation Computer Systems, 20(3):475-487, 2004.

[Schenk04] O. Schenk and K. Gartner. *On Fast Factorization Pivoting Methods for Sparse Symmetric Indefinite Systems*. Technical Report, Department of Computer Science, University of Basel, 2004, submitted.

[Sonn89] P. Sonneveld. *CGS, a Fast Lanczos-Type Solver for Nonsymmetric Linear Systems*. SIAM Journal on Scientific and Statistical Computing, 10:36-52, 1989.

[Young71] D.M.Young. *Iterative Solution of Large Linear Systems*. New York, Academic Press, Inc., 1971.

• Extended Eigensolver

[Polizzi09] E. Polizzi, *Density-Matrix-Based Algorithms for Solving Eigenvalue Problems*, Phys. Rev. B. Vol. 79, 115112, 2009.

[Polizzi12] E. Polizzi, *A High-Performance Numerical Library for Solving Eigenvalue Problems: FEAST Solver v2.0 User's Guide*, arxiv.org/abs/1203.4031, 2012.

[Bai00] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe and H. van der Vorst, editors, *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, 2000.

[Sleijpen96] G. L. G. Sleijpen and H. A. van der Vorst. *A Jacobi-Davidson iteration method for linear eigenvalue problems*. SIAM J. Matrix Anal. Appl., 17:401-425, 1996.

• VS

[AVX] Intel. *Intel® Advanced Vector Extensions Programming Reference*.

[Billor00] Nedret Billor, Ali S. Hadib, and Paul F. Velleman. *BACON: blocked adaptive computationally efficient outlier nominators*. Computational Statistics & Data Analysis, 34, 279-298, 2000.

[Bratley87] Bratley P., Fox B.L., and Schrage L.E. *A Guide to Simulation*. 2nd edition. Springer-Verlag, New York, 1987.

[Bratley88] Bratley P. and Fox B.L. *Implementing Sobol's Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software, Vol. 14, No. 1, Pages 88-100, March 1988.

[Bratley92] Bratley P., Fox B.L., and Niederreiter H. *Implementation and Tests of Low-Discrepancy Sequences*, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 195-213, July 1992.

[BMT] Intel. *Bull Mountain Technology Software Implementation Guide*.

[Coddington94] Coddington, P. D. *Analysis of Random Number Generators Using Monte Carlo Simulation*. Int. J. Mod. Phys. C-5, 547, 1994.

[Fritsch80] Fritsch, F. N and Carlson, R. E. *Monotone Piecewise Cubic Interpolation*. SIAM Journal on Numerical Analysis (SIAM) 17 (2): 238-246, 1980.

[Gentle98] Gentle, James E. *Random Number Generation and Monte Carlo Methods*, Springer-Verlag New York, Inc., 1998.

[Hyman83] Hyman, J. M. *Accurate monotonicity preserving cubic interpolation*, SIAM J. Sci. Stat. Comput. 4, 645-654, 1983.

[IntelSWMan] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 3 vols.

[L'Ecuyer94] L'Ecuyer, Pierre. *Uniform Random Number Generation*. Annals of Operations Research, 53, 77-120, 1994.

[L'Ecuyer99] L'Ecuyer, Pierre. *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*. Mathematics of Computation, 68, 225, 249-260, 1999.

[L'Ecuyer99a] L'Ecuyer, Pierre. *Good Parameter Sets for Combined Multiple Recursive Random Number Generators*. Operations Research, 47, 1, 159-164, 1999.

- [L'Ecuyer01] L'Ecuyer, Pierre. *Software for Uniform Random Number Generation: Distinguishing the Good and the Bad*. Proceedings of the 2001 Winter Simulation Conference, IEEE Press, 95-105, Dec. 2001.
- [Kirkpatrick81] Kirkpatrick, S., and Stoll, E. *A Very Fast Shift-Register Sequence Random Number Generator*. Journal of Computational Physics, V. 40. 517-526, 1981.
- [Knuth81] Knuth, Donald E. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.
- [Maronna02] Maronna, R.A., and Zamar, R.H., *Robust Multivariate Estimates for High-Dimensional Datasets*, Technometrics, 44, 307-317, 2002.
- [Matsumoto98] Matsumoto, M., and Nishimura, T. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, Pages 3-30, January 1998.
- [Matsumoto00] Matsumoto, M., and Nishimura, T. *Dynamic Creation of Pseudorandom Number Generators*, 56-69, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, <http://www.math.sci.hiroshima-u.ac.jp/%7Em-mat/MT/DC/dc.html>.
- [NAG] NAG Numerical Libraries. http://www.nag.co.uk/numeric/numerical_libraries.asp
- [Rocke96] David M. Rocke, *Robustness properties of S-estimators of multivariate location and shape in high dimension*. The Annals of Statistics, 24(3), 1327-1345, 1996.
- [Saito08] Saito, M., and Matsumoto, M. *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*. Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, Pages 607 – 622, 2008.
<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html>
- [Salmon11] Salmon, John K., Morales, Mark A., Dror, Ron O., and Shaw, David E., *Parallel Random Numbers: As Easy as 1, 2, 3*. SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011.
- [Schafer97] Schafer, J.L., *Analysis of Incomplete Multivariate Data*. Chapman & Hall, 1997.
- [Sobol76] Sobol, I.M., and Levitan, Yu.L. *The production of points uniformly distributed in a multidimensional cube*. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (In Russian).
- [SSL Notes] *Intel® oneMKL Summary Statistics Application Notes*, a document present on the Intel® oneMKL product at <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html>
- [VS Notes] *Intel® oneMKL Vector Statistics Notes*, a document present on the Intel® oneMKL product at <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html>
- [VS Data] *Intel® oneMKL Vector Statistics Performance*, a document present on the Intel® oneMKL product at <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html>

- **VM**

- [C99] ISO/IEC 9899:1999/Cor 3:2007. Programming languages -- C.

- [Muller97] J.M.Muller. *Elementary functions: algorithms and implementation*, Birkhauser Boston, 1997.
- [IEEE754] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-2008.
- [VM Data] *Intel® oneMKL Vector Mathematics Performance and Accuracy*, a document present on the Intel® oneMKL product at <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html>

• FFT

- [1] E. Oran Brigham, *The Fast Fourier Transform and Its Applications*, Prentice Hall, New Jersey, 1988.
- [2] Athanasios Papoulis, *The Fourier Integral and its Applications*, 2nd edition, McGraw-Hill, New York, 1984.
- [3] Ping Tak Peter Tang, *DFTI - a new interface for Fast Fourier Transform libraries*, ACM Transactions on Mathematical Software, Vol. 31, Issue 4, Pages 475 - 507, 2005.
- [4] Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.

• Optimization Solvers

- [Conn00] A. R. Conn, N. I.M. Gould, P. L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, MPS-SIAM Series on Optimization edition, 2000.

• Data Fitting Functions

- [deBoor2001] Carl deBoor. *A Practical Guide to Splines*. Revised Edition. Springer-Verlag New York Berlin Heidelberg, 2001.
- [Schumaker2007] Larry L Schumaker. *Spline Functions: Basic Theory*. 3rd Edition. Cambridge University Press, Cambridge, 2007.
- [StechSub76] S.B. Stechhkin, and Yu Subbotin. *Splines in Numerical Mathematics*. Izd. Nauka, Moscow, 1976.

For a reference implementation of BLAS, sparse BLAS, LAPACK, and ScaLAPACK packages visit www.netlib.org.

Glossary

A^H	Denotes the conjugate transpose of a general matrix A . See also conjugate matrix.
A^T	Denotes the transpose of a general matrix A . See also transpose.
band matrix	A general m -by- n matrix A such that $a_{ij} = 0$ for $ i - j > l$, where $1 < l < \min(m, n)$. For example, any tridiagonal matrix is a band matrix.
band storage	A special storage scheme for band matrices. A matrix is stored in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and <i>diagonals</i> of the matrix are stored in rows of the array.
BLAS	Abbreviation for Basic Linear Algebra Subprograms. These subprograms implement vector, matrix-vector, and matrix-matrix operations.

BRNG	Abbreviation for Basic Random Number Generator. Basic random number generators are pseudorandom number generators imitating i.i.d. random number sequences of uniform distribution. Distributions other than uniform are generated by applying different transformation techniques to the sequences of random numbers of uniform distribution.
BRNG registration	Standardized mechanism that allows a user to include a user-designed BRNG into the VSL and use it along with the predefined VSL basic generators.
Bunch-Kaufman factorization	Representation of a real symmetric or complex Hermitian matrix A in the form $A = PUDU^H P^T$ (or $A = PLDL^H P^T$) where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .
c	When found as the first letter of routine names, <i>c</i> indicates the use of the single-precision complex data type.
CBLAS	C interface to the BLAS. See BLAS.
CDF	Cumulative Distribution Function. The function that determines probability distribution for univariate or multivariate random variable X . For univariate distribution the cumulative distribution function is the function of real argument x , which for every x takes a value equal to probability of the event $A: X \leq x$. For multivariate distribution the cumulative distribution function is the function of a real vector $x = (x_1, x_2, \dots, x_n)$, which, for every x , takes a value equal to probability of the event $A = (X_1 \leq x_1 \ \& \ X_2 \leq x_2, \ \& \ \dots, \ \& \ X_n \leq x_n)$.
Cholesky factorization	Representation of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A in the form $A = U^H U$ or $A = L L^H$, where L is a lower triangular matrix and U is an upper triangular matrix.
condition number	The number $\kappa(A)$ defined for a given square matrix A as follows: $\kappa(A) = \ A\ \ A^{-1}\ $.
conjugate matrix	The matrix A^H defined for a given general matrix A as follows: $(A^H)_{ij} = (a_{ji})^*$.
conjugate number	The conjugate of a complex number $z = a + bi$ is $z^* = a - bi$.
d	When found as the first letter of routine names, <i>d</i> indicates the use of the double-precision real data type.
dot product	The number denoted $x \cdot y$ and defined for given vectors x and y as follows: $x \cdot y = \sum_i x_i y_i$. Here x_i and y_i stand for the i -th elements of x and y , respectively.
double precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $2.23 \cdot 10^{-308} < x < 1.79 \cdot 10^{308}$. For this data type, the machine precision ε is approximately 10^{-15} , which means that double-precision numbers usually contain no more than 15 significant decimal digits. For more information, refer to <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture</i> .
eigenvalue	See eigenvalue problem.

eigenvalue problem	A problem of finding non-zero vectors x and numbers λ (for a given square matrix A) such that $Ax = \lambda x$. Here the numbers λ are called the eigenvalues of the matrix A and the vectors x are called the eigenvectors of the matrix A .
eigenvector	See eigenvalue problem.
elementary reflector(Householder matrix)	Matrix of a general form $H = I - \tau v v^T$, where v is a column vector and τ is a scalar. In LAPACK elementary reflectors are used, for example, to represent the matrix Q in the QR factorization (the matrix Q is represented as a product of elementary reflectors).
factorization	Representation of a matrix as a product of matrices. See also Bunch-Kaufman factorization, Cholesky factorization, LU factorization, LQ factorization, QR factorization, Schur factorization.
FFTs	Abbreviation for Fast Fourier Transforms. See "Fourier Transform Functions".
full storage	A storage scheme allowing you to store matrices of any kind. A matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} stored in the array element $a(i, j)$.
Hermitian matrix	A square matrix A that is equal to its conjugate matrix A^H . The conjugate A^H is defined as follows: $(A^H)_{ij} = (a_{ji})^*$.
I	See identity matrix.
identity matrix	A square matrix I whose diagonal elements are 1, and off-diagonal elements are 0. For any matrix A , $AI = A$ and $IA = A$.
i.i.d.	Independent Identically Distributed.
in-place	Qualifier of an operation. A function that performs its operation in-place takes its input from an array and returns its output to the same array.
Intel® oneAPI Math Kernel Library (oneMKL)	Abbreviation for Intel® oneAPI Math Kernel Library.
inverse matrix	The matrix denoted as A^{-1} and defined for a given square matrix A as follows: $AA^{-1} = A^{-1}A = I$. A^{-1} does not exist for singular matrices A .
LQ factorization	Representation of an m -by- n matrix A as $A = LQ$ or $A = (L \ 0)Q$. Here Q is an n -by- n orthogonal (unitary) matrix. For $m \leq n$, L is an m -by- m lower triangular matrix with real diagonal elements; for $m > n$,

$$I = \begin{bmatrix} I_1 & \\ & I_2 \end{bmatrix}$$

where L_1 is an n -by- n lower triangular matrix, and L_2 is a rectangular matrix.

LU factorization

Representation of a general m -by- n matrix A as $A = PLU$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).

machine precision

The number ε determining the precision of the machine representation of real numbers. For Intel® architecture, the machine precision is approximately 10^{-7} for single-precision data, and approximately 10^{-15} for double-precision data. The precision also determines the number of significant decimal digits in the machine representation of real numbers. See *also* double precision and single precision.

MPI

Message Passing Interface. This standard defines the user interface and functionality for a wide range of message-passing capabilities in parallel computing.

MPICH

A freely available, portable implementation of MPI standard for message-passing libraries.

orthogonal matrix

A real square matrix A whose transpose and inverse are equal, that is, $A^T = A^{-1}$, and therefore $AA^T = A^T A = I$. All eigenvalues of an orthogonal matrix have the absolute value 1.

packed storage

A storage scheme allowing you to store symmetric, Hermitian, or triangular matrices more compactly. The upper or lower triangle of a matrix is packed by columns in a one-dimensional array.

PDF

Probability Density Function. The function that determines probability distribution for univariate or multivariate continuous random variable X . The probability density function $f(x)$ is closely related with the cumulative distribution function $F(x)$.

For univariate distribution the relation is

$$F(x) = \int_{-\infty}^x f(t) dt.$$

For multivariate distribution the relation is

$$F(x_1, x_2, \dots, x_n) = \int_{-\infty}^{x_1} \int_{-\infty}^{x_2} \dots \int_{-\infty}^{x_n} f(t_1, t_2, \dots, t_n) dt_1 dt_2 \dots dt_n.$$

positive-definite matrix

A square matrix A such that $Ax \cdot x > 0$ for any non-zero vector x . Here \cdot denotes the dot product.

pseudorandom number generator

A completely deterministic algorithm that imitates truly random sequences.

QR factorization

Representation of an m -by- n matrix A as $A = QR$, where Q is an m -by- m orthogonal (unitary) matrix, and R is n -by- n upper triangular with real diagonal elements (if $m \geq n$) or trapezoidal (if $m < n$) matrix.

random stream	An abstract source of independent identically distributed random numbers of uniform distribution. In this manual a random stream points to a structure that uniquely defines a random number sequence generated by a basic generator associated with a given random stream.
RNG	Abbreviation for Random Number Generator. In this manual the term "random number generators" stands for pseudorandom number generators, that is, generators based on completely deterministic algorithms imitating truly random sequences.
Rectangular Full Packed (RFP) storage	A storage scheme combining the full and packed storage schemes for the upper or lower triangle of the matrix. This combination enables using half of the full storage as packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels as the full storage.
s	When found as the first letter of routine names, <i>s</i> indicates the use of the single-precision real data type.
ScaLAPACK	Stands for Scalable Linear Algebra PACKage.
Schur factorization	Representation of a square matrix A in the form $A = ZTZ^H$. Here T is an upper quasi-triangular matrix (for complex A , triangular matrix) called the Schur form of A ; the matrix Z is orthogonal (for complex A , unitary). Columns of Z are called Schur vectors.
single precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $1.18 \times 10^{-38} < x < 3.40 \times 10^{38}$. For this data type, the machine precision (ϵ) is approximately 10^{-7} , which means that single-precision numbers usually contain no more than 7 significant decimal digits. For more information, refer to <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture</i> .
singular matrix	A matrix whose determinant is zero. If A is a singular matrix, the inverse A^{-1} does not exist, and the system of equations $Ax = b$ does not have a unique solution (that is, there exist no solutions or an infinite number of solutions).
singular value	The numbers defined for a given general matrix A as the eigenvalues of the matrix AA^H . See <i>also</i> SVD.
SMP	Abbreviation for Symmetric MultiProcessing. Intel® oneAPI Math Kernel Library (oneMKL) offers performance gains through parallelism provided by the SMP feature.
sparse BLAS	Routines performing basic vector operations on sparse vectors. Sparse BLAS routines take advantage of vectors' sparsity: they allow you to store only non-zero elements of vectors. See BLAS.
sparse vectors	Vectors in which most of the components are zeros.
storage scheme	The way of storing matrices. See full storage, packed storage, and band storage.
SVD	Abbreviation for Singular Value Decomposition. See <i>also</i> Singular value decomposition section in "LAPACK Auxiliary and Utility Routines".
symmetric matrix	A square matrix A such that $a_{ij} = a_{ji}$.

transpose	The transpose of a given matrix A is a matrix A^T such that $(A^T)_{ij} = a_{ji}$ (rows of A become columns of A^T , and columns of A become rows of A^T).
trapezoidal matrix	A matrix A such that $A = (A_1 A_2)$, where A_1 is an upper triangular matrix, A_2 is a rectangular matrix.
triangular matrix	A matrix A is called an upper (lower) triangular matrix if all its subdiagonal elements (superdiagonal elements) are zeros. Thus, for an upper triangular matrix $a_{ij} = 0$ when $i > j$; for a lower triangular matrix $a_{ij} = 0$ when $i < j$.
tridiagonal matrix	A matrix whose non-zero elements are in three diagonals only: the leading diagonal, the first subdiagonal, and the first super-diagonal.
unitary matrix	A complex square matrix A whose conjugate and inverse are equal, that is, that is, $A^H = A^{-1}$, and therefore $AA^H = A^H A = I$. All eigenvalues of a unitary matrix have the absolute value 1.
VML	Abbreviation for Vector Mathematical Library. See "Vector Mathematical Functions".
VSL	Abbreviation for Vector Statistical Library. See "Statistical Functions".
z	When found as the first letter of routine names, z indicates the use of the double-precision complex data type.

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Third Party Content

Intel® oneAPI Math Kernel Library includes content from several 3rd party sources that was originally governed by the licenses referenced below:

- Portions© Copyright 2001 Hewlett-Packard Development Company, L.P.
- Sections on the Linear Algebra PACKage (LAPACK) routines include derivative work portions that have been copyrighted:
 - © 1991, 1992, and 1998 by The Numerical Algorithms Group, Ltd.
- Intel® oneAPI Math Kernel Library supports LAPACK 3.5 set of computational, driver, auxiliary and utility routines under the following license:

Copyright © 1992-2011 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.

Copyright © 2000-2011 The University of California Berkeley. All rights reserved.

Copyright © 2006-2012 The University of Colorado Denver. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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

- The original versions of the Basic Linear Algebra Subprograms (BLAS) from which the respective part of Intel® oneAPI Math Kernel Library was derived can be obtained from <http://www.netlib.org/blas/index.html>.
- XBLAS is distributed under the following copyright:

Copyright © 2008-2009 The University of California Berkeley. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.

- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- The original versions of the Basic Linear Algebra Communication Subprograms (BLACS) from which the respective part of Intel® oneAPI Math Kernel Library was derived can be obtained from <http://www.netlib.org/blacs/index.html>. The authors of BLACS are Jack Dongarra and R. Clint Whaley.
- The original versions of Scalable LAPACK (ScaLAPACK) from which the respective part of Intel® oneAPI Math Kernel Library was derived can be obtained from <http://www.netlib.org/scalapack/index.html>. The authors of ScaLAPACK are L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley.
- The original versions of the Parallel Basic Linear Algebra Subprograms (PBLAS) routines from which the respective part of Intel® oneAPI Math Kernel Library was derived can be obtained from http://www.netlib.org/scalapack/html/pblas_qref.html.
- PARDISO (PARallel DIrect SOLver)* in Intel® oneAPI Math Kernel Library was originally developed by the Department of Computer Science at the University of Basel (<http://www.unibas.ch>). It can be obtained at <http://www.pardiso-project.org>.
- The Extended Eigensolver functionality is based on the Feast solver package and is distributed under the following license:

Copyright © 2009, The Regents of the University of Massachusetts, Amherst.
Developed by E. Polizzi
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Some Fast Fourier Transform (FFT) functions in this release of Intel® oneAPI Math Kernel Library have been generated by the SPIRAL software generation system (<http://www.spiral.net/>) under license from Carnegie Mellon University. The authors of SPIRAL are Markus Puschel, Jose Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo.
- Open MPI is distributed under the New BSD license, listed below.

Most files in this release are marked with the copyrights of the organizations who have edited them. The copyrights below are in no particular order and generally reflect members of the Open MPI core team who have contributed code to this release. The copyrights for code used under license from other parties are included in the corresponding files.

Copyright © 2004-2010 The Trustees of Indiana University and Indiana University Research and Technology Corporation. All rights reserved.

Copyright © 2004-2010 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.

Copyright © 2004-2010 High Performance Computing Center Stuttgart, University of Stuttgart. All rights reserved.

Copyright © 2004-2008 The Regents of the University of California. All rights reserved.

Copyright © 2006-2010 Los Alamos National Security, LLC. All rights reserved.

Copyright © 2006-2010 Cisco Systems, Inc. All rights reserved.

Copyright © 2006-2010 Voltaire, Inc. All rights reserved.

Copyright © 2006-2011 Sandia National Laboratories. All rights reserved.

Copyright © 2006-2010 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.

Copyright © 2006-2010 The University of Houston. All rights reserved.

Copyright © 2006-2009 Myricom, Inc. All rights reserved.

Copyright © 2007-2008 UT-Battelle, LLC. All rights reserved.

Copyright © 2007-2010 IBM Corporation. All rights reserved.

Copyright © 1998-2005 Forschungszentrum Juelich, Juelich Supercomputing Centre, Federal Republic of Germany

Copyright © 2005-2008 ZIH, TU Dresden, Federal Republic of Germany

Copyright © 2007 Evergrid, Inc. All rights reserved.

Copyright © 2008 Chelsio, Inc. All rights reserved.

Copyright © 2008-2009 Institut National de Recherche en Informatique. All rights reserved.

Copyright © 2007 Lawrence Livermore National Security, LLC. All rights reserved.

Copyright © 2007-2009 Mellanox Technologies. All rights reserved.

Copyright © 2006-2010 QLogic Corporation. All rights reserved.

Copyright © 2008-2010 Oak Ridge National Labs. All rights reserved.

Copyright © 2006-2010 Oracle and/or its affiliates. All rights reserved.

Copyright © 2009 Bull SAS. All rights reserved.

Copyright © 2010 ARM Ltd. All rights reserved.

Copyright © 2010-2011 Alex Brick . All rights reserved.

Copyright © 2012 The University of Wisconsin-La Crosse. All rights reserved.

Copyright © 2013-2014 Intel, Inc. All rights reserved.

Copyright © 2011-2014 NVIDIA Corporation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.

- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- The Safe C Library is distributed under the following copyright:

Copyright (c)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

- HPL Copyright Notice and Licensing Terms

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed at the University of Tennessee, Knoxville, Innovative Computing Laboratories.
4. The name of the University, the name of the Laboratory, or the names of its contributors may not be used to endorse or promote products derived from this software without specific written permission.